MASTER'S THESIS

A flexible mechanism to provide for a refactoring advice based on the source code entity

Verduin, E.

Award date:
2021

Evert Verduin

A FLEXIBLE MECHANISM TO PROVIDE FOR A REFACTORING ADVICE BASED ON THE SOURCE CODE ENTITY

Thesis Presentation: Friday july 23, 2021 at 10:00 AM.

Open Universiteit
de beste! www.ou.nl

A FLEXIBLE MECHANISM TO PROVIDE FOR A REFACTORING ADVICE BASED ON THE SOURCE CODE ENTITY

by

Evert Verduin

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University of the Netherlands,
Faculty of Science
Master's Programme in Software Engineering
to be defended publicly on Friday july 23, 2021 at 10:00 AM.

Student number:		
Course code:	IMA0002	
Thesis committee:	mw. dr. ir. Sylvia Stuurman (Chair),	Open University
	dhr. dr. ir. Harrie Passier (Primary Supervisor),	Open University
	dhr. prof. dr. Lex Bijlsma (Supervisor),	Open University.

ACKNOWLEDGEMENTS

I consider this research project as an adventurous journey with big ups and downs. The big ups bring enjoyment and provide motivation to carry on the research, while the big downs set you back and force you to think in different, and often unknown directions. The big downs do not necessarily motivate to carry on, but they are of utmost importance to complete the journey successfully.

I found that during these big downs the motivation was often provided externally, by my family. My joyful kids, Teun, Gijs, and Koen, thank you for making me smile whenever it was necessary. My wife, Sarah, thank you for always supporting me during the entire process. My father, Henk Verduin, thank you for motivating me to carry on. My mother Tinie Verduin, which I am still thankful for teaching me how to chase, and how to realize my dreams.

On the professional level, I would like to thank my fellow researchers, Herman Hilberink and William Wernsen. The enjoyment of our numerous Saturday morning "refactorbuddies" sessions will echo along. Without our sessions, the knowledge on this subject would never be on the level as where it is now!

Finally, I would like to thank my two supervisors, dr. ir. Harrie Passier, and prof. dr. Lex Bijlsma. Whenever I was stuck or took the wrong turn, they provided just the right push to get on track again. The feedback that I received from my supervisors helped to get my research and this thesis more complete.

Evert Verduin
Leiden, juni 2021

CONTENTS

List of Definitions

1

SUMMARY

Refactoring as a process can positively influence the maintainability of a software product, however, refactoring software can be considered as a hazardous activity. To recognize the hazards involved in the refactoring process requires specialistic skills, experience, and knowledge. An automated refactoring guidance tool could assist in training these skills and competencies.

The current refactoring process consists of performing refactoring steps which are concluded by compile-and-test steps to detect introduced errors. These introduced errors can be caused by invisible hazards which need to be understood and recognized. How can the impact of these underlying, and invisible hazards be reduced?

To reduce the impact of these hazards, the refactoring steps should be made smaller. These smaller steps, called microsteps, will have a lower impact and will have fewer hazards. As a consequence, the verification of the compile-and-test stage can no longer be performed. How can this compile-and-test stage be replaced?

A new mechanism replaces the compile-and-test stage and provides an advice. Software harm can occur when a microstep is performed in combination with the presence of a hazardous code fragment. Each microstep can have multiple hazardous code fragments and each hazardous code fragment can have numerous variations. How can hazardous code fragments, and their variations, be described in a flexible, understandable, and reusable manner?

A new query language, which is based on the Abstract Syntax Tree (AST) representation of the source code, has been developed to solve this problem. This query language uses hazardous code pattern descriptions to accurately detect potential software harm in a program under refactoring. How can a customized refactoring advice be generated to prevent software harm during refactoring activity?

An advice mechanism is designed to collects facts from the source code entity. These collected facts are provided by detectors that use the hazardous code pattern descriptions. By using this method, a very detailed advice is provided to the refactorer.

The concepts of this study are validated by a pen and paper exercise. These results show that the provided abstract solution works.

2

SAMENVATTING

Refactoring als proces kan de onderhoudbaarheid van een software product positief beïnvloeden, echter moet het refactoren van software wel worden gezien als een gevaarlijke activiteit. Om deze gevaren, welke betrokken zijn bij het refactoring proces, te kunnen herkennen zijn specialistisch vaardigheden, ervaring en kennis benodigd. Een automatisch refactoring tool zou ondersteuning kunnen bieden om deze ervaring en kennis en trainen.

Het huidige refactoring proces bestaat uit het uitvoeren van refactoring stappen welke worden afgesloten met compile-en-test stappen om zo te kunnen controleren of er fouten zijn geïntroduceerd. Deze geïntroduceerde kunnen worden veroorzaakt door meerdere onderliggende- en vaak onzichtbare gevaren die begrepen en onthouden moeten worden. Hoe kan de impact van deze onderliggende, en onzichtbare gevaren worden verkleind?

Om de impact van deze gevaren te verkleinen, moeten de refactoring stappen worden verkleind. Deze kleinere stappen, genaamd microstappen, zullen een kleinere impact en minder gevaren hebben. Echter, een consequentie van deze microstappen is dat de verificatie van de compile-en-test stap niet meer kan worden uitgevoerd. Hoe kan deze compile-en-test stap worden vervangen?

Een nieuw mechanisme vervangt de compile-en-test stap en geeft vervolgens een advies. Software beschadiging is een fenomeen dat kan optreden als een microstap wordt uitgevoerd in combinatie met de aanwezigheid van een gevaarlijk code fragment. Hoe kunnen gevaarlijke code fragmenten worden beschreven op een flexibele, begrijpelijke en herbruikbare manier?

Een nieuwe querytaal, welke gebaseerd is op de abstracte syntax boom (AST) representatie van de bron code moet dit probleem gaan oplossen. Deze querytaal gebruikt gevaarlijke-code-patroon-beschrijvingen om zo nauwkeurig software beschadiging te kunnen detecteren. Hoe kan een op maat gemaakt advies worden gegenereerd om zo software beschadiging tijdens refactoring te voorkomen?
Een adviesmechanisme welke feiten uit de broncode verzamelt is hiervoor ontworpen. Deze verzamelde feiten worden geleverd door detectoren die de gevaarlijke-code-patroon-beschrijvingen gebruiken. Door deze methode te hanteren, kan er een zeer gedetailleerd advies aan diegene die de refactoring uitvoert worden verstrekt.

De concepten van deze studie zijn gevalideerd met een pen-en-papier opdracht. De resultaten hiervan tonen aan dat deze abstracte oplossing werkt.

3

INTRODUCTION

There is high market pressure to produce software products as fast as possible and fix issues with the software product later. Maintaining sufficient software quality combined with ongoing changing product requirements forces a higher focus on the software maintenance process[Fuggetta and Di Nitto, 2014].

The maintainability of a software product can be positively influenced by a process called refactoring. This process has been described as the refactoring activities in a study by Tourwé and Mens [Tourwé and Mens, 2004]. These activities comprehend:

- The identification of where the software should be refactored.
- Which refactoring(s) should be applied to these identified places.
- A guarantee that the applied refactoring preserves behavior.
- Application of the refactoring.
- An assessment of the effect which the refactoring had on the quality characteristics of the software.
- Maintaining consistency between the refactored program code and other software artifacts.

An important activity in this list of activities is the guarantee of preserving behavior in the third step. Preserving program behavior means that the initial functionality of a program, after refactoring must be maintained [Tourwé and Mens, 2004]. However, before program behavior preservation guarantees can be given, a proper diagnostic step should be performed. This important diagnostic step is missing from their research.

The diagnostic step could aid the refactorer in detecting software hazards that are induced by their refactoring actions. These hazards comprehend the chance of introducing errors to the software. The current refactoring processes, like the one described by Fowler, avoid hazardous refactoring situations. Understanding these hazardous refactoring situations is a difficult task since it requires multiple software engineering knowledge on an expert level. Without this expert-level knowledge, it is much safer to choose not to perform hazardous refactoring, which implies that some software parts could remain to be unrefactored. Code that remains unrefactored could suffer from architectural decay and a decreased software maintainability.

An example of a hazard that is detected in the diagnostic phase of a refactoring, is when a parameter will be removed from a Java method. A specific hazard that coincides with such action is, that callers, which attempt to call the changed method, will no longer work. In this case, the compiler will flag that these callers have a parameter too much in their parameter list. However fixing this, by changing all related callers could introduce a cascade of new issues. Like dynamic binding issues via the superclass and siblings of that subclass that became non-functional and all due to a single deletion of one of the parameters from a method definition. When this hypothetical situation occurs in real program code, that program code has become harmed. Detection of these refactoring induced hazards can be quite complicated and it requires:

- Intensive knowledge about the programming language.
- Thorough knowledge about the source code and its interrelations.
- An awareness of code combinations that can cause issues.

These competencies and skills can be trained by learning software refactoring on the job, with the help of a refactoring expert. However, to remain compliant with the ever-expanding software market, more and more refactoring experts are needed. An automated refactoring guidance tool seems a promising educational alternative to this high demand for refactoring experts. Besides the educational aspect, an automated refactoring guidance tool could aid the professional programmer too, since more and more existing software needs to be refactored nowadays. Considering that this existing software is often developed by others, a diagnostic refactoring tool could certainly help the programmer during the complex refactoring process. Some Integrated Development Environment (IDE)'s like eclipse already offer some sort of automated refactoring mechanism, which performs refactorings as a one-step fire-and-forget action. These tools could ease the refactoring process, but the lack of a proper diagnostic process and the lack of a possibility to execute the refactoring step-by-step are big drawbacks.

A study by Patrick de Beer, showed that an automated tool could help as guidance during software refactoring. A specifically tailored tool was composed out of a so-called Refactoring Advice Graph (RAG) and addressed a small and testable set of refactorings. His work creates a vast understanding of the automated refactoring guidance- process and aspects in general [Beer, 2019]. However, a concern on the usage of the RAG's used in his tool is that these are difficult to understand and it is therefore complicated to produce new RAG's for other refactorings.

Our contribution will be to find a method that provides refactoring advice deduced from a diagnostic step. This method has to be flexible, simple, reusable, self-explanatory and it should elaborate on the functional part of the work done by Patrick de Beer. The diagnostic step should provide the facts necessary to provide useful refactoring advice for

the refactorer. An example of an advice could be to skip a refactoring, because in the source code under refactoring a fragment of code has been detected that is hazardous in combination with the current performed refactoring action. The harm to the software, which is caused if the advice is neglected, is a program preservation break failure. Besides advice, a possible alternative to the above-described situation should be offered, but it should be kept in mind that an alternative is not always possible. To provide refactoring advice, facts have to be collected from the entire problem context, which includes the source code under refactoring, the refactoring steps with their inflicted problems, and a list of code combinations that cause issues for that refactoring-step inflicted problem. The contributions this research will provide are:

- *A transformation of Fowler's refactoring mechanic steps into microsteps.*
- *A list of hazards per microstep.*
- *A list of each of these hazards accompanied with their hazardous code pattern descriptions.*
- *A flexible method or recipe to detect hazardous code fragments in a source code entity.*
- *A method to combine separately detected software harm into a final composite advice.*
- *A validation of the concept.*

In this thesis chapter 4, will describe the related work. Chapter 5 describes the first analysis. Chapter 6 discusses the research questions and chapter 7 will present the research results. Chapter 8 will present the conclusion and is followed by chapter 9, which presents the recommendations and future work.

4

RELATED WORK

The work described in this chapter is meant to elaborate on the current research on refactoring guidance and on the refactoring process itself. This section will also provide missing links or gaps in the existing literature that prevent the current known refactoring processes to provide the necessary diagnostic step as described in the introduction. The first section will focus on the work on automated refactoring guidance done by Patrick de Beer, the second section will focus on the work done by Martin Fowler and the third section will focus on the work done by William Opdyke.

4.1. AUTOMATED GUIDANCE IN THE REFACTORING PROCESS

Patrick de Beer presented an automated refactoring tool that uses a feed-forward mechanism to generate a refactoring advice based on the code context and with the help of a so-called RAG. This RAG is a directed graph used as a decision tree that combines code context, code-property detectors, and advice templates to generate the refactoring advice. However, each refactoring requires a dedicated RAG, so there is very little future reuse. The more complex the refactoring becomes, the more complex the accompanied RAG will be. The prototype presented in the study worked well for a specific refactoring however, the increasing complexity of the RAG's is a concern. [Beer, 2019]. Despite the drawbacks of the RAG, the tool demonstrated that a refactoring advice can be provided on a feedforward basis with an automated tool.

4.1.1. PROGRAMMING PROBLEMS SOLVED IN THE RAG

To prevent the automated refactoring tool from ending up in a deadlock, the software implementation demands that all branches in the RAG must be conjugated by a negated branch [Beer, 2019]. This eases the software implementation, but it will make the RAG solution more complex. This added complexity makes it difficult to expand the RAG since it will lead to a strong increment of empty advice branches, as can be seen in the empty advice nodes in figure 4.1. Another restriction is that each node in the RAG must provide advice, even when no advice is necessary, which is the reason for these empty advice-nodes in the RAG. This bounds the RAG's to a specific refactoring and minimizes its reuse.

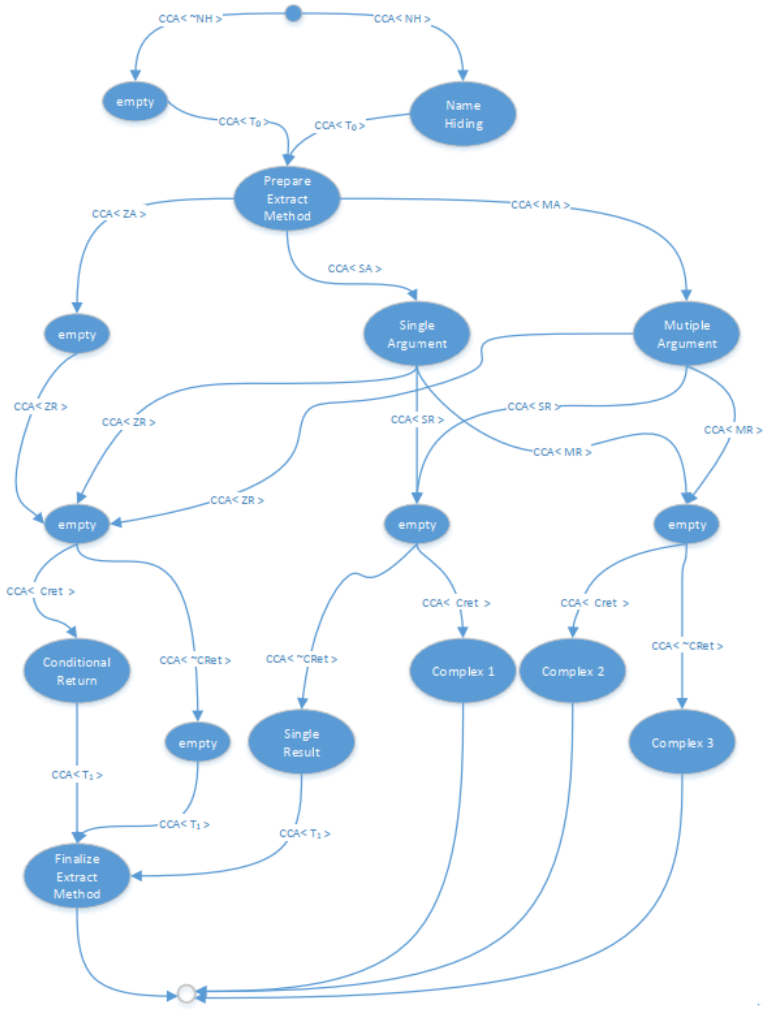

Figure 4.1: An extract-method RAG

4.1.2. No recipe for generating new refactorings

Another concern is the lack of a proper recipe to construct these RAG's, which makes it complicated to understand existing RAG's and makes it even more complicated to produce new RAG's for other refactorings. The study done by Patrick de Beer does not provide a solid base to recreate a proper recipe for creating new RAG's for other refactorings.

4.2. Refactoring as a process described by Martin Fowler

Martin Fowler eased the refactoring process by introducing refactoring mechanics as a step-by-step recipe, with some minimal diagnostics to perform specific refactorings. However, the majority of these mechanics contain vague directives, where instinct is a necessary ingredient. This refactoring-instinct, mentioned by Fowler, is not a clear definition of skill. Fowler states that experienced refactorers will make the right decision instinctively, which implicates that refactoring experience is a necessity [Fowler, 1999].

4.2.1. VAGUE DIRECTIVES IN THE MECHANICS BY FOWLER

Fowler's mechanics sometimes contain vague directives and are multi-interpretable. Fowler states that these vague directives often concern difficult refactoring situations with difficult decisions to make. These vague descriptions can therefore not be stated as exact steps within the mechanics. An example of these vague directives can be found in the inline method mechanics, figure 4.2, where the mechanics are presented in a simple and straightforward step-by-step way. However, a remark concludes that these mechanics are not easy to perform and can provide lots of difficulties that are not described and that if you run into these undefined complexities, you should not perform this refactoring. Another refactoring-mechanic of interest can be found in appendix B where the "Add a parameter to method" mechanic is multi-interpretable. When wrongly interpreted, these mechanics will not be program behavior preserving. As a consequence refactoring activity could fail, due to these directives. Therefore it is important that the directives in Fowler's mechanics are well described and that they are unambiguous.

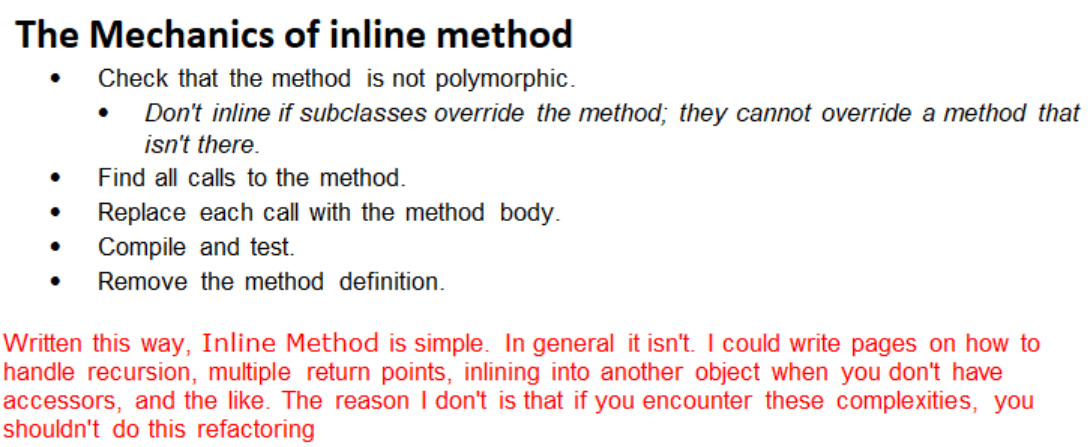

Figure 4.2: Fowler's mechanics for inline method

4.2.2. UNADDRESSED HAZARDS IN THE REFACTORING PROCESS

Refactoring software involves the chance of introducing errors, which makes refactoring a complicated and hazardous activity. Recognizing these hazards demands expert knowledge and refactoring experience during refactoring activities. From current literature, it is unclear what specific hazards are involved with certain refactorings. Often, as an alternative in the refactoring mechanics, the solution offered is to stop and avoid the current refactoring activity. The complication of these unaddressed hazards is that refactoring efforts could eventually lead to unsuccessful refactoring attempts, by introducing errors in the program code. These unsuccessful attempts combined with the complexity of the refactoring process will probably negatively affect the willingness to perform refactorings in the future. The awareness for refactoring hazards can be increased by accurately specifying what hazards are involved with certain refactorings.

4.3. PROGRAM BEHAVIOR PRESERVATION AS DESCRIBED BY WILLIAM OPDYKE

A study by Opdyke explained that specific preconditions of refactorings can ensure program behavior preservation. To proceed with automated refactoring, the preconditions must be met. If one of the preconditions of that refactoring is not met, program behavior preservation for that specific refactoring cannot be guaranteed and the refactoring should be avoided [Opdyke, 1992].

4.3.1. PROGRAM BEHAVIOR PRESERVATION DETECTION BY USING SPECIFIC PRE-CONDITIONS FOR REFACTORINGS

A troublesome refactoring can be seen in figure 4.3, where a rename method refactoring is performed on a function F2 in the subclass. The function F2 will be renamed to function F1. After the refactoring, function F3 will call the newly declared function F1 in the local class, instead of the function F1 from the superclass. Which leads to a break in program behavior after the refactoring took place.

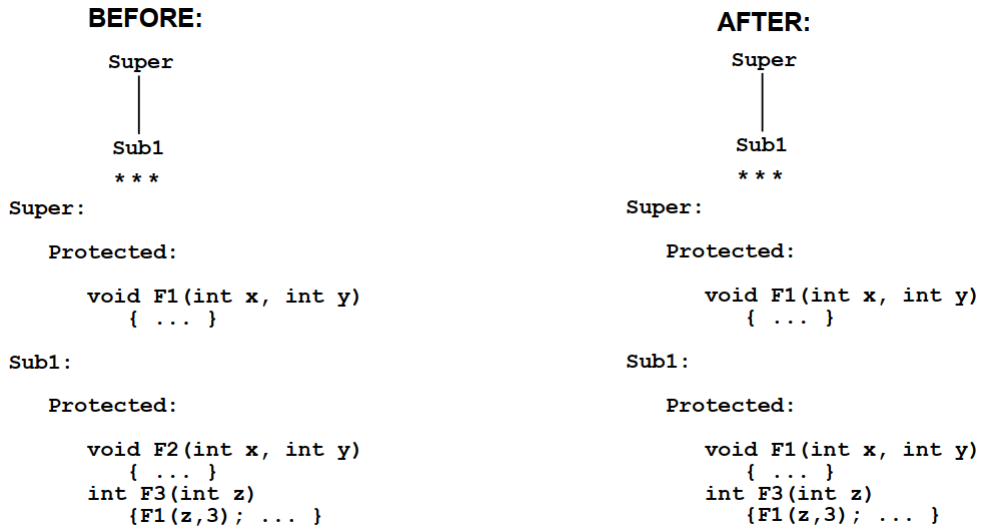

Figure 4.3: The erroneous renaming of member function F2 [Opdyke, 1992]

An example of a precondition that would not have been met before this troublesome refactoring takes place, is the pre-condition below in equation 4.1. To simplify this example, the assumption is made that F1 is the new name of the method.

$$\forall\, member \in commonSuperclass.locallyDefinedMemberFunctions, member.name \neq F1.name. \tag{4.1}$$

This pre-condition will not be met when it is applied to the example from figure 4.3, since the name of function F1 in the superclass and the new name of function F2 (which will be F1) is the same. In this example, only one precondition is used for demonstrational purposes, but it would take a list of preconditions to test for the behavior preservation of the example in figure 4.3 [Opdyke, 1992]. Instead of following Opdyke's suggestion to skip a troublesome refactoring, a tailored solution in the form of an advice should be offered

instead. With enough insight and a proper advice, troublesome refactorings which should be skipped according to Opdyke, could be performed without issues.

5

FIRST ANALYSIS

This section provides new insights on the subjects and will provide information that is needed to improve the comprehensibility of this subject. These insights are artifacts of the performed study and are not found in the literature. This section will elaborate on the issues with the hazards of refactoring and how the impact of refactoring can be decreased. A diagnostic process will be introduced, which establishes the terms and definitions that will be used in this study. This section will also elaborate on the cause of the hazards induced by refactoring activity.

5.1. LOWERING THE IMPACT OF REFACTORING ACTIVITY

Big composite, or coarse-grained, refactoring steps have a higher negative impact in case of problems and vice versa. The current size of the refactoring steps provided by Fowler's mechanics is not small enough. There is a need for smaller, fine-grained refactoring steps with less impact compared to the more coarse-grained refactoring steps from Fowler's mechanics. Since each step in the mechanics must be program behavior preserving, the majority of Fowler's mechanic steps cannot be made smaller than they currently are. With current coarse-grained refactoring steps, a mechanic step can comprehend multiple hazards. For an accurate and more precise description of hazards, there is a need for smaller and more fine-grained refactoring steps.

5.1.1. REDUCING THE GRANULARITY OF THE MECHANIC REFACTORING STEPS TO MAKE A MORE PRECISE DESCRIPTION OF HAZARDS POSSIBLE

With the current coarse-grained granularity of Fowler's mechanic refactoring steps, an accurate description of hazards is difficult since the hazard granularity is also coarse-grained. In the case of an error, it becomes difficult to isolate the root cause of the problem, since a single problem can have multiple causes involved. Fowler's mechanics steps should prefer-

ably be divided into smaller steps with lower impact, and these step should produce code which is still understandable and human-readable.

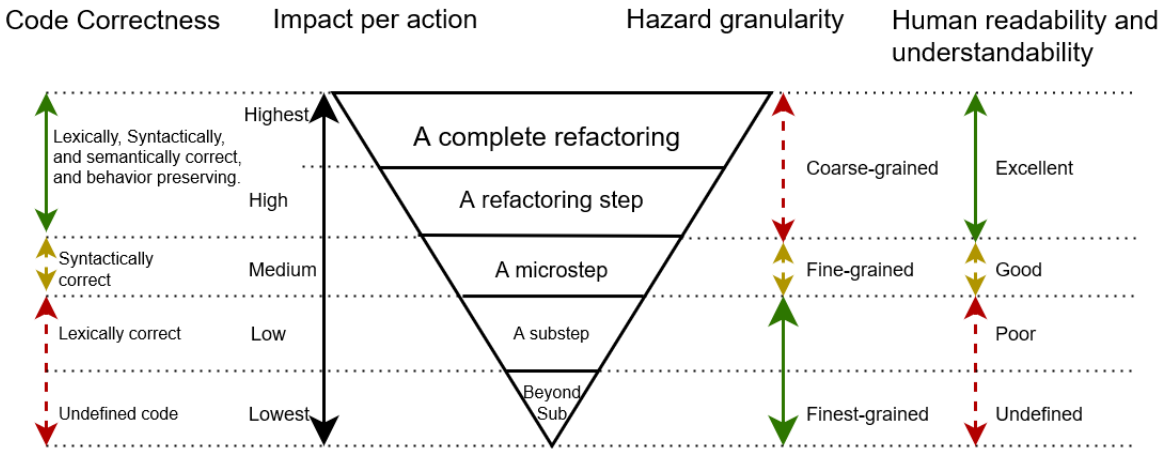

Figure 5.1: Anatomy of refactoring steps, its code correctness, its impact, its hazard granularity, its readability, and its understandability

Figure 5.1 presents a breakdown of possible refactoring steps with varying step size. The granularity of these refactoring steps is derived from the model in figure 5.2, which represents the block diagram of a generic compiler that produces byte-code.

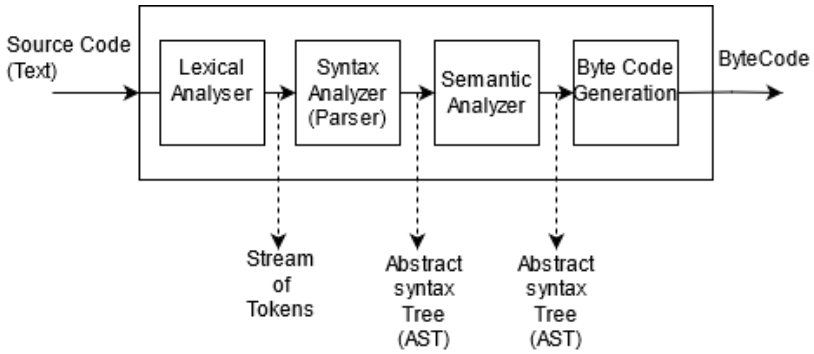

Figure 5.2: A block diagram, with an intermediate output description of the elements of a compiler that produces bytecode.

This study used the stages of the compiler to define the predicates for the source code under compilation. These predicates are:

- **Behavior Preserving.** Code that can be compiled without problems. This code is readable, understandable, and does not need correction to successfully compile. This code is lexically, syntactically, and semantically correct and this code has unchanged behavior. Which means that the behavior of the code is maintained.
- **Semantical correct.** Code that can be compiled without problems. This code is readable, understandable, and does not need correction to successfully compile. This code is lexically, syntactically, and semantically correct.
- **Syntactical correct.** This code is readable, understandable, and might need some semantical correction to successfully compile. This code is lexically, and syntactically correct.

- **Lexical correct.** This code could contain syntactical errors and/or semantic errors. This code is lexically correct and might need syntactical and semantic corrections to compile.
- **Undefined code.** Not lexically correct "code" can represent anything, like random sequences of characters.

To further explain what these predicates mean, examples of errors are given in Table 5.1 below.

Error types	Example	Comments
Semantical error	int i="string";	int and string are not compatible
Syntactical error	int i	Colon is forgotten
Lexical error	int void=0;	void is a reserved keyword.
Undefined code	*anything*	This can be anything, like random characters, as long as the lexical scanners do not detect tokens.

Table 5.1: Error types with code example and comments

5.1.2. MICROSTEPS

If performed correctly, Fowler's refactoring mechanics are semantically correct and can be tested by a successful compilation. If the refactoring mechanics are performed correctly, these steps do not introduce lexical, syntactical, and/or semantic errors and are behavior preserving. If Fowler's refactoring steps need to be divided into smaller steps, it is logical that the size of these steps follows the predicates from the model in 5.2. If they would not, then the new smaller refactoring steps, even in compositions, would yield code that cannot be compiled nor tested. Refactoring steps that produce lexically correct code can still be syntactically incorrect on their own, which is undesirable. Refactoring steps that produce syntactically correct code, can still be semantically incorrect. An example, if a microstep adds a method to a specific scope where a similar method is already present, then this is syntactically correct but semantically incorrect. However, the subsequent microsteps should provide for semantical correctness. Therefore, if composed correctly from Fowler's refactoring mechanics, the correct composition of microsteps will produce semantically correct code. Modifying software, by performing microsteps, can lead to problems with the software, like the introduction of errors.

Definition 5.1.1

A microstep: A small refactoring step that produces syntactically correct code, but it is not behavior preserving. This smaller refactoring step is a derivate of Fowler's mechanic refactoring steps. The correct composition of one or multiple microsteps replaces the mechanic's steps from Fowler where the impact of the composite refactoring remains the same. However, since the microsteps on their own have a much smaller impact, the hazard definition is less compound per performed refactoring action. A microstep yields code that can be successfully parsed. After parsing, an Abstract Syntax Tree (AST) is produced. Performing a microstep on a source code under refactoring is a hazardous activity.

5.1.3. TRANSFORMING FOWLERS MECHANICS INTO MICROSTEPS

A microstep is similar to a step in the refactoring mechanics. A microstep can act on a subject, where the action can either be adding, removing, or renaming. The subject can be either a class, an attribute, a method, an interface, or a parameter. However, a microstep is a much smaller step, with less impact than a refactoring mechanic step. Checks like "compile and test" and other tests are excluded. A table with possible microsteps is presented in Table 5.2 below.
Without transforming the refactoring mechanics into microsteps, a proper diagnostic process that provides insight into the induced hazards will be very difficult. By using microsteps, the program behavior preservation for these introduced microsteps can no longer be guaranteed, but the granularity of the microsteps makes an accurate and precise definition of these hazards possible.

Microstep	Class	Attribute	Method	Interface	Parameter
Add					
Remove					
Rename					

Table 5.2: Microsteps with subjects. This table is supposed to be empty.

A typical refactoring step from Fowler's mechanics consists out of multiple microsteps. An example is the refactoring mechanic step "Rename Attribute". A microstep transformation of this mechanic step would consist out of two steps: "Remove Attribute" and "Add Attribute." Both "Remove Attribute" and "Add Attribute" microsteps have their own set of associated hazards, which were previously combined into one composite mechanic refactoring step. This makes the definition of possible hazards for each microstep less difficult.

5.2. DEFINITION OF THE DIAGNOSTIC PROCESS

Since the diagnostic process in software refactoring is a new concept, clear and harmonized terms that define the diagnostic process are missing from the literature. This research requires clear and harmonized terms to define the diagnostic process. To help to harmonize and to define these terms and definitions, the ISO-14971 is adopted. This standard has its origin as a descriptive definition of risk-management terms for medical devices [ISO-14971, 1998] and the terms defined in this harmonization are assumed to apply to the diagnostic process used in this research. Therefore the terms and definition of this standard will be used and will be applied in a software refactoring context. We isolate the four main terms from the ISO-14971 harmonization:

- *Hazard* - Something that has the potential to cause harm.
- *A hazardous situation* - A circumstance that exposes an entity to one or more hazards.
- *Harm* - Damage that is caused to an entity as a consequence of a hazard.
- *Risk* - The probability of the occurrence of harm and the severity of that harm.

The following sections will now address these important new terms, which are applied to the software refactoring process.

5.2.1. SOFTWARE HAZARD

Changes made to software have the potential to cause harm to software. When changes are made the chance that bugs or errors are introduced is increased and it may lead to software hazard. These changes could lead to software hazards like non-working software or even software defects. Even using the software can be hazardous when a user deviates from the intended use of a piece of software. An example: A software tester that is seeking for bugs or errors deviates from the intended use. The concerning hazard in this case, is that the software fails due to a bug or error. An example, driving a car is also considered a hazard and may lead to harm. Driving a car in the dark, tired, and without glasses is a cascade of hazards that increases the chance of harm. A safe and hazards free situation has a low amount of manageable hazards, while a safe situation without any hazard is fairly impossible.

Typical software hazards are:

- **Adding features** to the software that is not open for extension.
- **Deviation of use** Software that is being used to its original intent. ie. using software concurrently while it was not intended for concurrent use.
- **Refactoring** software. Rewriting software to change its maintainability.
  - **Performing a microstep** within a refactoring. Performing microsteps on a code under refactoring can be a hazardous activity.

Refactoring software is an activity where a hazard is introduced. Tracing bugs is detecting hazards that are already present in the software in the form of bugs. Therefore, differentiation between software hazards must be made:

- **Hazard that is already present in the software**, like software with bugs or software that does not meet the open-closed principle.
- **Hazard that is introduced to the software** when a refactoring is performed and the hazard was not present before the refactoring activity.

Definition 5.2.1

A software hazard: presents a risk or threat to the software due to an activity that is associated with the use of software, or that is associated with making changes to the software. Activities like refactoring introduce new hazards while adding features to the software that is not open for extension reveals already present hazards in that software.

5.2.2. SOFTWARE HARM

When software becomes damaged, software harm occurs. Damaged software needs repair to restore its original functionality. Software harm has a severity level and a type.

Example of **severity levels** are:

- **Severity** can be stated as:
  - No severity, like no error.
  - Minor, like introducing overloading in a class or compiler warnings.
  - Low, like introducing a compiler and/or runtime error.
  - Major, like introducing program behavior preservation break.

Examples of **types of harm** that can be caused to the software are:

- **Inconveniences** like compiler warnings that do not cause issues. No severity.
- **Errors** in the software that prevent the software from a successful compilation or errors that prevent the software from entering its running state. This is a low severity.
- **Software defects** like program behavior preservation breaks. This is the case when the software has unintended different behavior or functionality after the refactoring activity. This is a major severity.

An example: when the microstep remove method is performed on a code under refactoring and callers to that method are still present and unaware of the removal of the method. Performing this specific microstep on this code under refactoring will damage the software. The harm caused will be a compile error with low severity.

Definition 5.2.2

Software harm: Damage that is done to the software, like the introduction of compiler warnings, errors, or software defects.

5.2.3. HAZARDOUS CODE PATTERNS

Figure 5.3 represents a concrete example of a hazardous code pattern, together with the code context and the refactoring subject. The hazardous code pattern in this example is a fragment of statements and definitions from the code of a concrete program. These concrete patterns will be referred to as (concrete) hazardous code fragments. Although the word "fragment" is not 100% descriptive, it is close enough.
The refactoring subject is the single subject that is being refactored by a microstep (underlined in the figure). This subject can be found in table 5.2 on the horizontal axis. The code context is the entire code and everything associated with the code to make it functional. The hazardous code fragment consists out of multiple subjects (printed boldly in the figure) that can have relations with each other. The relationship between the subjects of a hazardous code fragment is not directly visible and needs effort to comprehend.
Example: methodB in the superclass is a subject that is a part of the hazardous code fragment, but it is in another class.
An important notice, subjects that are part of the hazardous code fragment do not have to be in the close vicinity of the refactoring subject. In the hazardous code fragment in figure 5.3 the subjects with their relationships form a pattern, although this pattern is not directly visible from this representation.

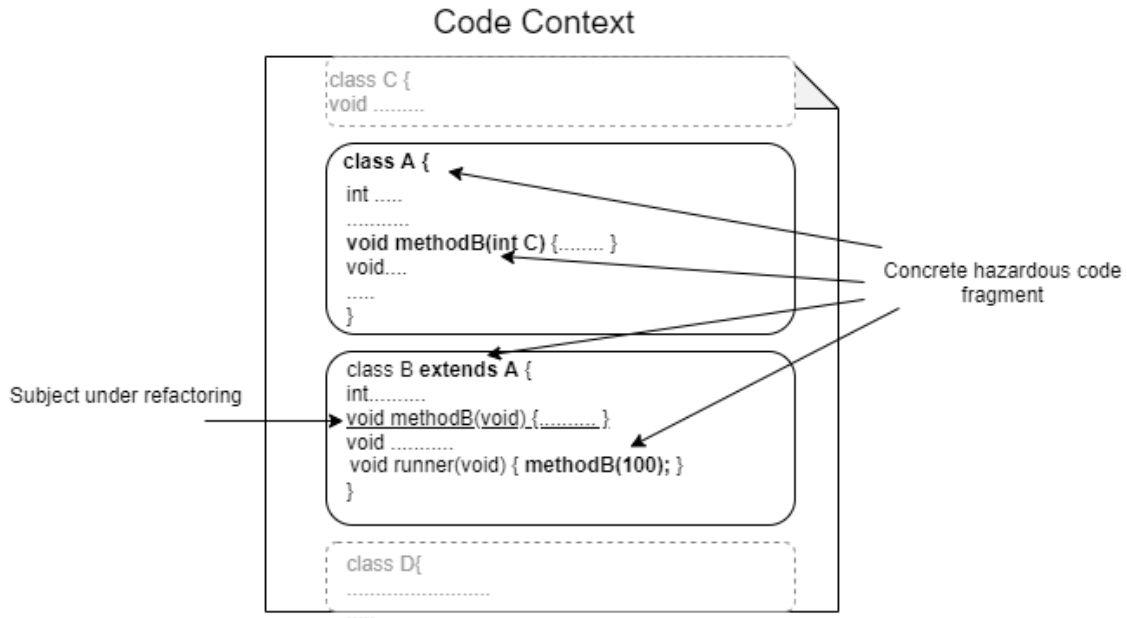

Figure 5.3: A refactoring subject, the code context, and a concrete hazardous code pattern

The presence of other code in the code context, in figure 5.3, is not of interest to the description of this specific hazardous code fragment. This hazardous code fragment concerns the fact that there is a child class B of a superclass A, which has a method call to a method B with a parameter of the type int which is declared in the superclass A. If the microsteps remove method (subject methodB(void) and add method (subject methodB(int)

is performed on the subject under refactoring in figure 5.3 then software harm can occur.

To detect a (concrete) hazardous code fragment in program code, a description of a pattern consisting out of specific code characteristics that describe all possible tolerated variations on these concrete hazardous code fragments is necessary. An example of tolerated variations that can occur in these hazardous code fragments is: varying class names, varying method names, or code that is in between statements, etc. An example of an intolerable variation can be seen in figure 5.3, which is the location of the caller MethodB(100) residing in the subclass B. These patterns consist of specific code characteristics that describe hazardous code fragments that will be referred to as hazardous code pattern descriptions. Microsteps have their own unique and limited list of hazards and each hazard has its list of hazardous code pattern descriptions. The hazardous code pattern descriptions will have to match all tolerated varieties of code fragments. Therefore, a method to create hazardous code pattern descriptions that describe all tolerable variations of concrete hazardous code fragments in a generic code context is necessary.

Definition 5.2.3

Hazardous code fragments: The hazardous code fragment is a fragment of statements and definitions from the code of a concrete program. These statements and definitions are facts that are collected from the source code entity.

Definition 5.2.4

Hazardous code pattern descriptions: A description of a pattern that consists of specific code characteristics that describes all possible tolerated variations on a concrete hazardous code fragment. These hazardous code pattern descriptions are necessary to detect hazardous code fragments in a generic source code entity.

5.2.4. Potential software harm

The probability that harm is caused to software depends on the right circumstances. These circumstances can be defined as trigger-events. Trigger-events consist of a sequence or combinations of events. Since the behavior of software is similar to discrete mathematics, the occurrence of harmful-trigger-events follows a binary behavior. In our research, harmful-trigger-events are defined as a pair that consists out of a microstep and its accompanying hazardous code pattern description.

Definition 5.2.5

Harmful-trigger-event: a pair that consists out of a microstep and its accompanying hazardous code pattern description.

The occurrence of these harmful-trigger-events can be very accurately measured by retrieving them from the software and refactoring actions. If an occurrence of a harmful-trigger-event is detected, then harm is considered to be present. If its presence is not measured, no induced harm is considered to be present.
The formula for software harm can therefore be written as:

$$SoftwareHarm = Presence(harmful\_trigger\_event) \times Severity \tag{5.1}$$

- **Presence(harmful_trigger_event)** depends on:
  - The discrete presence (yes/no) of a harmful-trigger-event. A harmful-trigger-event, a microstep, and its accompanying hazardous code pattern description.

Definition 5.2.6

Potential software harm: The product of the discrete presence of an harmful-trigger-event and the severity of the harm caused by this trigger-event. The presence of a harmful-trigger-event has a binary representation, which is either zero or one. If a harmful-trigger-event is detected (1), software harm will happen and the harm will be equal to the severity. If a harmful-trigger-event is not detected (0), the induced harm will be equal to zero, which is no harm.

5.2.5. Describing software harm which is induced by refactoring activity

Fowler's mechanic steps form a guideline through the entire refactoring process. All steps in that refactoring process are initiated and directed from that guideline. Transforming these mechanics steps into microsteps, helps to define the accompanying harm of a microstep more precisely. It should be intuitively known that some microsteps like the "delete method" microstep have the potential to cause software harm. By experiment, it became known that some microsteps are prone to cause software harm when applied to a specific hazardous code fragment. Therefore, it is important to notice that it is the microstep that initiates the software hazard. The pair existing out of a microstep and its accompanying hazardous code pattern description forms the harmful-trigger-event. Without knowing all the harmful-trigger-events that can cause software harm, refactoring will remain a hazardous exercise. To reduce the chance of causing software harm during refactoring activity, the presence of hazardous code fragments, which is corresponding to the performed microstep, needs to be detected in the source code entity. To prioritize the software refactoring risks, the severity of the harm needs to be specified as well as can be seen in formula 5.1.

To describe and to detect potential software harm there is a need for:

- **The harmful-trigger-event** consisting of:
  - **A microstep** - The microstep that could cause software harm.
  - **A description of a hazardous code pattern description that accompanies the hazard of the microstep** - A hazardous code pattern description, so that the hazardous code fragment can be detected in the source code entity.
- **The potentially caused software harm** consisting out of:
  - **The type of software harm caused**. What type of harm is caused, like a compiler warning.
  - **The severity of that software harm caused**. What is the severity of the harm, like having unused variables after refactoring.

5.2.6. DESCRIBING CONCRETE HAZARDOUS CODE FRAGMENTS WITH HUMAN LANGUAGE

To detect software harm, during a refactoring step, the hazardous code pattern description of a hazardous code fragment should be known. The purpose of these hazardous code pattern descriptions is to detect hazardous code fragments in the source code entity. During this research, the presence of numerous hazardous code fragments was manually detected in several code examples. Multiple hazardous code fragments were successfully identified and attempted to be transformed into hazardous code pattern descriptions. However, describing hazardous code fragments in natural human language is difficult. Transforming these hazardous code fragments into hazardous code pattern descriptions is even more challenging.

As a consequence of not being able to create hazardous code pattern descriptions, it becomes difficult or even impossible to recognize hazardous code fragments in the source code entity. The consequence of not being able to recognize these hazardous code fragments is that the detection of potential harm during refactoring will be difficult or even impossible. To perform a proper diagnostic step where the potential harm can successfully be detected, a method to describe hazardous code patterns and how to detect hazardous code fragments is necessary.

5.2.7. WHY A DESCRIPTION OF CONCRETE CODE FRAGMENTS IN HUMAN LANGUAGE IS DIFFICULT

To describe the problem of describing code fragments in human language, a simple code example is presented in figure 5.4 which will be used in the code fragment descriptions below. Since this is a code description, which is not hazardous (yet), it will be referred to as a code fragment.

```
public class ClassX
	{
	Public void testMeX(int justaparamX)
		{
		// empty body
		}
	}

public class ClassY extends ClassX
	{
	public void testMeY(int justaparamY)
		{
		// empty body
		}
	}
```

Figure 5.4: A simple code example with inheritance

There is no standard, nor literature which explains how to describe code fragments. During experimental exercises, experience was gained in how code fragments can be described in a structured way. An attempt with normal sentences to describe the code fragment is presented below:

"(1)In a situation where there are two public classes from which one is (2) a superclass with name ClassX and the other is the subclass of ClassX with the name ClassY and (3) there is a public method with the name testMeX in ClassX which does not return a value and that has an empty body. (4) There is also a public method with the name testMeY in ClassY that

does not return a value and that accepts a parameter of type int (6) and that has an empty body."

This attempt to describe the code fragment is difficult to comprehend and is too specific. For a new attempt, observe the listed version of the code fragment description below:

1. There are 2 classes in total.
2. There is a public superclass with the name ClassX and there is a public subclass of ClassX with the name ClassY.
3. ClassX contains one public method with the name testMeX which accepts a parameter of the type int and the method does not return data.
4. ClassY contains one public method with the name testMeY which accepts a parameter of the type int and the method does not return data.
5. Method testMeX has an empty body.
6. Method testMeY has an empty body.

Both examples describe the same entities and the relations between entities, in both examples the subjects are linked together by the numbers 1 to 6. In these descriptions, the overall situation is described in (1), which describes that there are 2 classes. Then in (2), the classes and their relationships with each other are described. In (3) and in (4), the entities inside the classes are described and in (5) and (6), the methods which have empty bodies are described.

Although both descriptions correctly describe the same example, it was found by experiment that the listed version of the code fragment description is much easier to comprehend than the sentenced version. The reason for this effect is that the listed example describes the hierarchy of the code fragment vertically. Understanding the relation of the lowest entity with the other entities means reading upwards, while the sentenced version of the description requires memorizing, sketching, and repetitive rereading. Second, the sentenced version is suffering from language constraints as well. The sentences must be grammatically correct, which is a major drawback. The listed version also suffers from language constraints, but because the sentences are much smaller, this drawback has less effect on comprehensibility. It is important to retrieve a method that can describe the relations between the entities of a source code fragment and one that is not bound by language constraints.

6

RESEARCH METHOD

6.1. MAIN RESEARCH GOAL

The focus of this study is to find a flexible method that provides a refactoring advice based on the performed refactoring activity and the source code under refactoring. The refactoring activities consist out of performing microsteps, while the advice will be based on the possible software harm that can be caused by the performed microstep and its accompanying hazardous code fragment.

The main research question: *How can we provide refactoring advice based on: the performed refactoring activity, and the source code under refactoring?*

6.2. DESCRIBING HAZARDOUS CODE FRAGMENTS

Describing hazardous code fragments in natural language is difficult, therefore a new comprehensible method to describe hazardous code fragments needs to be designed. Which will be the first research question of our study:

RQ1: *How can a hazardous code fragment be described in an easy to comprehend manner?*

6.2.1. APPROACH

To solve the issue of describing the hazardous code fragments, a proprietary and abstract query language is developed. This language forms the foundation for the other research questions and is bottom-up designed with previously addressed requirements, like flexibility, comprehensibility, and reuse build-in.

6.2.2. VALIDATION

The concept of this study, including the answers of RQ1, will be validated with a pen and paper exercise, which will be presented in this thesis in chapters 7.3.3 and 7.3.4.

6.3. MICROSTEP INDUCED HAZARD

Induced hazards should be specified per microstep. During refactoring, possible software harm is caused when a harmful-trigger-event is detected in the source code entity. To make it possible to detect possible software harm, there is a need for a list of hazardous code pattern descriptions which is accompanied by each hazard that a microstep has. It is important to notice that during the performance of a microstep, positive detection of the accompanying harmful-trigger-event for that microstep, will lead to software harm. So, a harmfull-trigger-event should be specified by the harm which it causes. A microstep on its own will be a hazardous activity, so it is logical to specify the hazards per microstep. Which is the basis for the second two research questions:

RQ2: *How can we transform Fowler's refactoring mechanic steps into microsteps?*

RQ2.1: *What will the hazard(s) of a particular microstep be?*

RQ2.2:*Which hazardous code pattern descriptions are needed for each microstep's induced hazard to detect potential software harm?.*

6.3.1. APPROACH

A large number of refactorings described by Fowler were used to retrieve the majority of these hazards. The mechanics were first translated into microsteps, since the hazards need to be specified on a microstep level. Next, the microstep induced hazards were used to retrieve the hazardous code pattern descriptions. The hazardous code pattern descriptions are described with the use of the query language from RQ1.

6.3.2. VALIDATION

The concept of this study, including the answers of RQ2, RQ2.1, and RQ2.2 will be validated with a pen and paper exercise, which will be presented in this thesis in chapters 7.3.3 and 7.3.4.

6.4. A FLEXIBLE DETECTION MECHANISM TO DETECT FOR POTENTIAL SOFTWARE HARM

This research will focus on providing a flexible mechanism to achieve the detection of potential software harm during refactoring activity. The mechanism will also provide an advice with suggestions to solve an issue. The mechanism will avoid hard-wired solutions. The outcome of RQ1 will provide a steady basis for the provided mechanisms in RQ3, where flexibility, comprehensibility, expandability, and reuse are important parameters that are designed bottom-up into RQ1. This leads us to the third and final research questions:

RQ3: *How can hazardous code fragments be detected in a source code entity flexibly?*

RQ3.1:*How can multiple detected potential software harm be combined into final composite refactoring advice?*

6.4.1. APPROACH

For RQ3, the hazardous code pattern descriptions with their aliases are used to detect hazardous code fragments. Further, the query language has been modified to communicate what entities it has detected. The purpose of the added communication feature is to provide final composite refactoring advice. The abstraction of the query language easily allows for these enhancements.

6.4.2. VALIDATION

The concept of this study, including the answers of RQ3, and RQ3.1 will be validated with a pen and paper exercise, which will be presented in this thesis in chapters 7.3.3 and 7.3.4.

6.5. SCOPE OF THE STUDY

For this study, I will only focus on the programming language Java. Static analysis methods will be used, which means that polymorphism is excluded from this research. Further, generic types, dynamic binding, and java reflection will also be excluded from this study and some of these exclusions will be marked as future work.

7

RESEARCH RESULTS

7.1. DESCRIBING CODE FRAGMENTS

When describing a single subject of a source code fragment solely, all relations with all other subjects have to be described. In other words, describing a single subject requires a description of the entire code context that is involved with that subject. By using another representation of the code fragment, the relations between the subject of the code fragment become much more clearer. The AST representation was found to be useful material to be studied for this purpose. The AST represents the hierarchy of a source code (fragment) where the code is represented by edges and nodes in a tree.

7.1.1. DESCRIBING THE RELATIONS OF CODE CONSTRUCTS IN A SOURCE CODE FRAGMENT WITH THE HELP OF AN AST REPRESENTATION

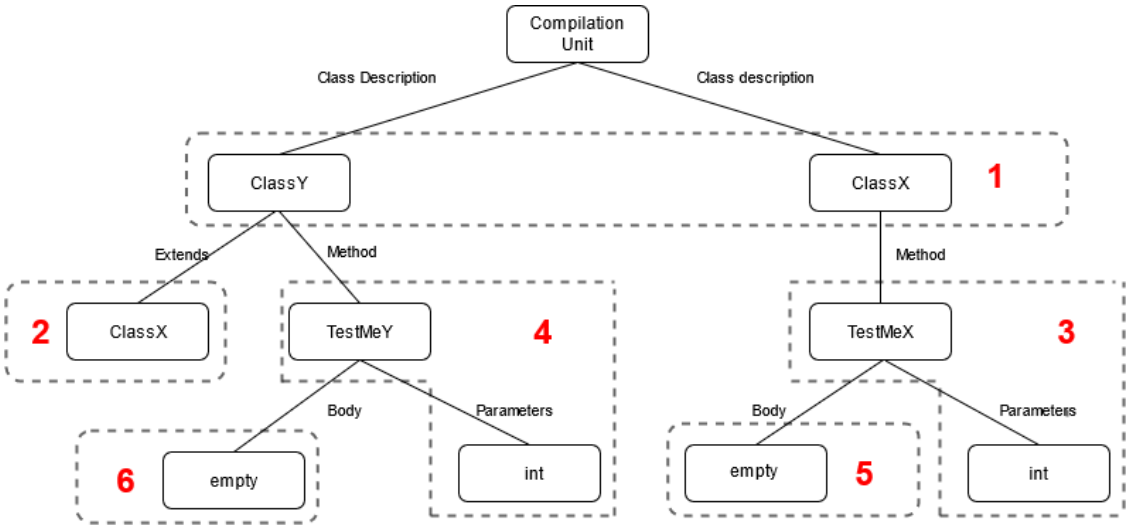

Figure 7.1: A simplified abstract syntax tree representation of the code from the example in figure 5.4

A highly simplified AST representation of the code example in figure 5.4 is presented in figure 7.1. This figure uses boxes with dashed lines and a number in red which repre-

sents the attempted description of the code example in chapter 5.2.7. The exact numbered descriptions, from chapter 5.2.7 are repeated below:

1. There are 2 classes in total.
2. There is a public superclass with the name ClassX and there is a public subclass of ClassX with the name ClassY.
3. ClassX contains one public method with the name testMeX which accepts a parameter of the type int and the method does not return data.
4. ClassY contains one public method with the name testMeY which accepts a parameter of the type int and the method does not return data.
5. Method testMeX has an empty body.
6. Method testMeY has an empty body.

The full-blown AST representation of this code fragment can be found in Appendix C in figure 10 and is too large to be used as an example, therefore some details in figure 7.1 are highly simplified. Details like modifiers, return types, method parameter names are left out of this example for simplification. The relations between the subjects in figure 7.1 are represented by lines/edges with a written description to simplify this representation. The nodes describe code constructs from the source code, while the edges describe the structure between these code constructs. ie. a code construct is a method declaration in a source code, like the construct void methodname(void). These code constructs will be further referred to as subjects when used in an AST context. Examples of subjects in an AST representation are the definition of a class, a method, a field, etc. When a subject with a relation is present in the simplified AST it automatically implicates that the code construct is present in the source code as well. The edge between the subject ClassX and the subject TestMeX in the AST representation describes the presence of a method TestMeX in Class ClassX in the source code and so on. The hierarchy is presented in a top-down manner. The subject with the highest hierarchy is the compilation unit. Second, are the classes and so on. Subjects on the same horizontal level share the same hierarchy level. The advantage of an AST representation is that the inner relational structure of a code fragment is quickly and easily revealed. The disadvantage of an AST representation is that it hides the functionality of the code fragment.

Definition 7.1.1

A subject: An node in an AST representation that can have a relation with other nodes or which is a start node or a leaf. This node represents a code construct in the source code.

Definition 7.1.2

A relation between subjects: The edges in an AST represent the relations between the subjects. This relation represents the relation between code constructs in the source code.

7.1.2. Describing scoping in a source code fragment with the use of an AST representation

The scope in an AST representation is defined as the region where subjects can be bound to an entity. That entity can be another subject, however the downward hierarchy of an AST applies. So, there is no upward scope in an AST representation. An example of a scope definition in an AST representation is the scope of the class with the name "ClassY". This scope comprehends all subjects and relations below that class subject. Another example is the scope of the method with the name "TestMeY" with comprehends all subjects and relations below that method subject.

Definition 7.1.3

A scope: The scope in an AST representation is defined as the region where subjects can be bound to an entity. That entity can be another subject, however, the downward hierarchy of an AST representation applies.

7.1.3. Abstracting an AST representation towards a query language

To accurately describe code fragments, the previously described findings need to be transformed into a new query language that is based on an abstraction of the AST representation. Since the relations are already described by the edges of an AST representation there is no need to additionally describe these.

Subjects

The subjects can be described by specifying their name, ie. "TestMeY" or "TestMeX", as represented in figure 7.2. It is important to mention that these subjects refer to the nodes in an AST representation and not the code fragment itself. A problem arises when both the classes "ClassY" and "ClassX" have methods with a similar name, only using the subject's name does not provide a unique reference to the correct subject in an AST representation.

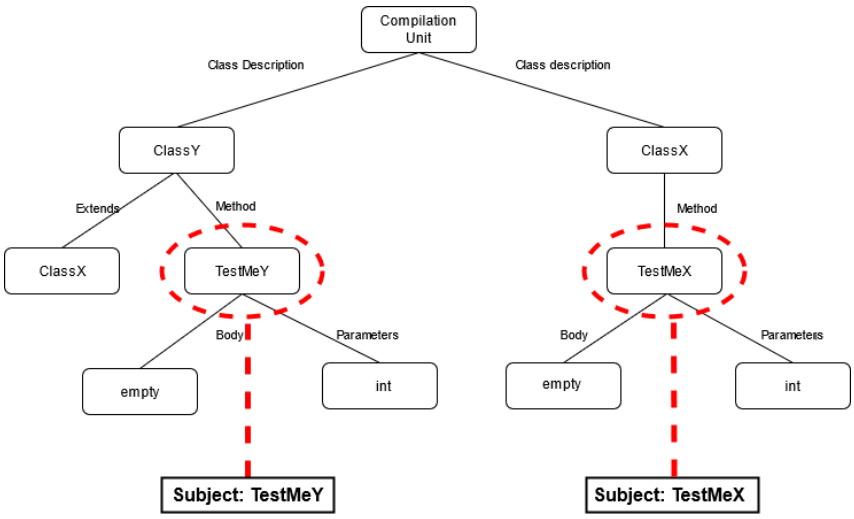

Figure 7.2: A simplified abstract syntax tree representation with subjects

As can be seen in figure 7.2, the subjects "TestMeY" and "TestMeX" are "visible" respectively in the scope of the node with the name "ClassY" and the node with the name "ClassX."

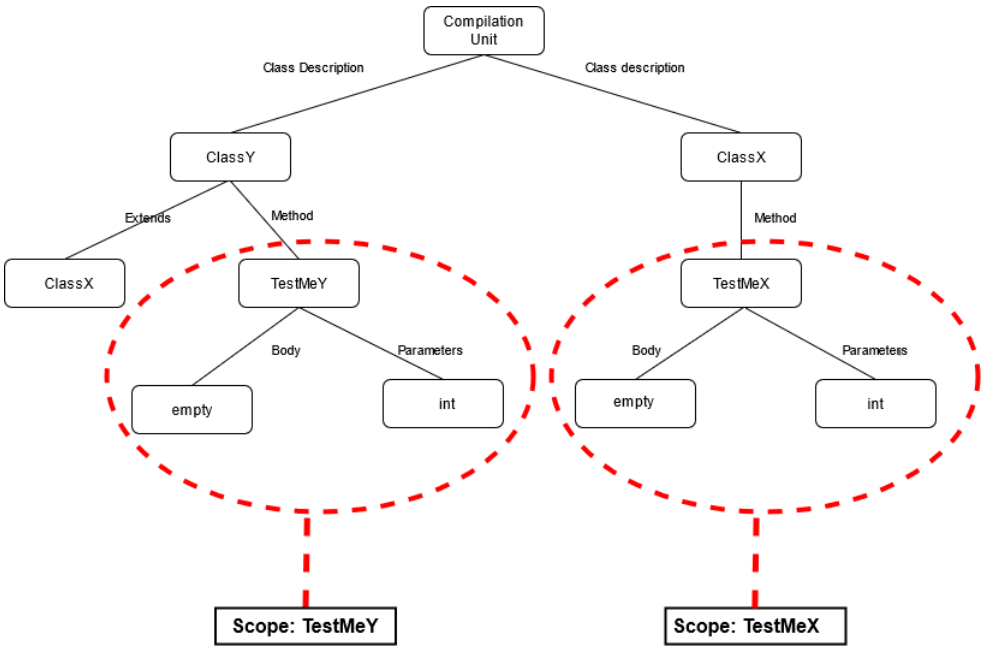

Figure 7.3: A simplified abstract syntax tree representation with scoping

SCOPE

In figure 7.3 two scopes can be observed, the "TestMeY" scope and the "TestMeX" Scope. When combining both the scope and the subject description, an exact and unique description is established. In example 7.4 a unique reference to the correct subjects can be seen. Describing the subject "TestMe" with scope "compilation unit" will refer to two subjects in the AST in this example. A subject description can yield more than one subject in an AST representation, which is not always desirable. In these situations, the scope is used together with the subject's name. An exact reference to the method "TestMe" in the class with the name "ClassY" will be referred to as subject "TestMe" in the scope of subject "ClassY". If a scope can represent a collection of subjects, then specifying the subject together with a scope represents the intersection of that subject with the collection of subjects that are determined by that scope.

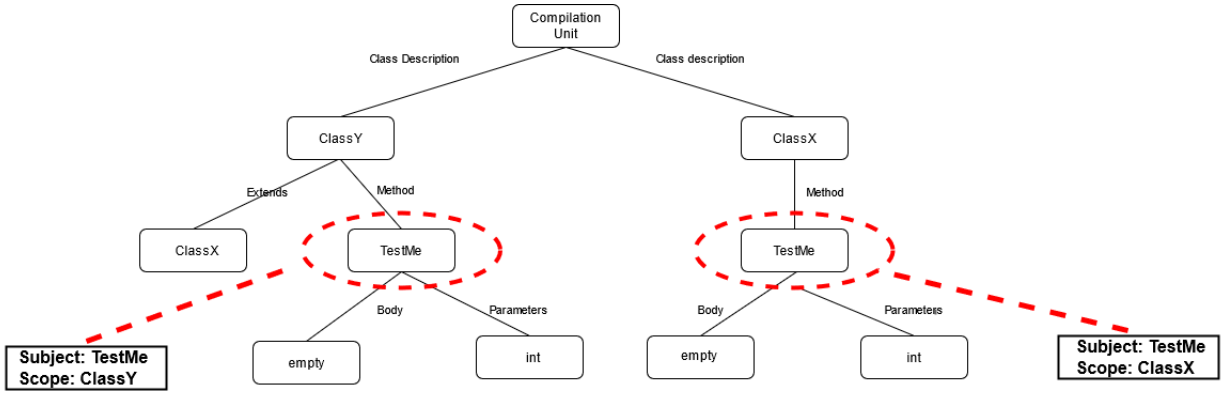

Figure 7.4: A simplified abstract syntax tree representation with subjects and their scope

Statements to describe subjects and scopes

The newly defined query language will make it possible to describe scopes and subjects in an intuitive, short, and comprehendible manner. These descriptions reference the position of subjects and/or scopes in an AST representation. An example of a statement is, method "TestMe" with scope "ClassY", which yields the node of method "TestMe" in the subclass "ClassY" in the AST representation in figure 7.4. These descriptions of subjects, scopes, and other newer elements in an AST representation will be further referenced as (query language) statements. Statements produce a reference to an AST entity, which can be perceived as an output.

Definition 7.1.4

A (query) statement: A description of entities in an AST representation, and which yields subjects, scopes, or other entities.

The output of a statement

The high level of abstraction of the query language provides for the necessary freedom to make any design choice. The example of the previous section displayed that the statement of the subject "TestMe" with scope "Compilation Unit" will refer to two subjects in the AST representation in figure 7.4. One could think of a situation where a statement will yield no subject or a situation where a statement will yield multiple subjects. An important design choice is that a statement can yield nothing, one entity, or multiple entities. An example of a statement's output is the node "TestMe" in the scope of class "ClassY".

Definition 7.1.5

The output of a statement: A statement yields a list of entities in an AST representation which represents no entity, one entity, or multiple entities.

PIPELINING OF STATEMENTS

When statements yield output, it should be possible to cascade separate statements to enhance the readability of the statements. Another important effect of cascading statements is that the reuse of statements will become possible. It is the reference to nodes in the AST that is communicated, which is a design choice that makes recursion, and real-time assessment of the pseudo-AST elements possible. These references can then be used for further examination by the next statement in the pipeline. An important design choice (implementation) is that the information output of each detector should be transparently transmitted through, or beside a detector. This transparent stream of information is necessary to delay the decision-making process to the last possible moment [Stuurman, 2015].

Definition 7.1.6

Pipelining of statements: A statement can be used as input for another statement. Statements can "communicate" via their produced output. The purpose of pipelining is to increase the comprehensibility and to increase the reusability of statements.

STATEMENT ALIASES

Since statements must be easy to comprehend and reuse of statements should be stimulated, statements can be referenced with an alias. An alias is a name that can be freely chosen, but the name may not contain spaces. An alias refers to the output of a statement, which is a list with no, one, or multiple entities in an AST representation. It is a design choice to make an alias refer to the output of a statement and not the statement itself.

Definition 7.1.7

Statement aliases: An alias is a name that can be freely chosen, without spaces. An alias refers to the output of a statement, which is a list with no, one, or multiple entities of an AST representation.

7.1.4. REDEFINITION OF THE AST REPRESENTATION

The AST representation in figure 10 in appendix C demonstrates that the original AST representation is too complicated and too big to be used as a model for the new query language. The query language used in this study will define a new abstraction on the AST representation to make the language and the representation easy to comprehend. Another advantage is that the language and the model can easily be modified, extended, and adapted because it acts as a facade in between the implementation. The abstraction on the AST representation that this study uses will be referred to as the pseudo-AST representation. To ensure that the correct predefined subjects are used in the query language, a reference to all known subjects will be presented in this section. Although this reference for subjects is provided as a tree in a hierarchical order, it should just be interpreted as a pick-list reference. If an element is not in the presented tree, it cannot be used as a subject in the query language statements. A remark is that the presented subjects can be changed to adapt to future changes.

INTERFACES AND ABSTRACT CLASSES

Figure 7.5 represents a tree of subjects that can be used in the query language. These subjects comprehend interfaces and (abstract) classes. A subject can have other subjects and/or leaves. By referencing the class subject in the pseudo-AST representation, the content of the leaves and other nodes of this subject can be accessed just like with an iteration. An example is that if the class subject is known, the name of that class can be easily retrieved since it is a property of the top-level subject Class. The end-nodes or leaves of the presented tree contain information and are always represented as a list. For example, the end-node "Name" will contain a list with one name, which is the name of that class. The "field" end-node will contain a list of all present fields in that class. Some end-nodes contain a reference to other subjects in the pseudo-AST representation. It is important to mention that these end-nodes always contain a list, even if there is just one value possible. The end-node "Super" will contain one or no reference to a superclass, so it will exist out of a list with one or no value. This design choice eases the use of the query language and its implementation since there is just one object type to work with. A default leaf does not have to be specified in query language expressions. If a class is referred, then the name end-node is used per default. The pseudo-AST representational subjects will not be described one by one, since these are programming language concepts and widely known, however, some subjects will be explained:

- Name - The name of this class. This end-node or leaf is the default.
- Sub - A reference to the subclass of this class, if set, this class is a superclass. If not set, then this class is not a superclass. This end-node holds a reference to the subclass in the pseudo-AST representation.
- Methods - A list of pseudo-AST representational references to all methods in that class.
- Fields - A list of pseudo-AST representational references to all fields in that class.
- Parent - A reference to the parent of that class. Which can be another class, which in that case means that the current class is an inner class. This could also be a package.
- Super - A reference to the superclass of this class, if set, this class is a subclass. If not set, then this class is not a subclass. This end-node holds a reference to the superclass in the pseudo-AST representation.
- Implements - The list of references to the implemented interfaces.

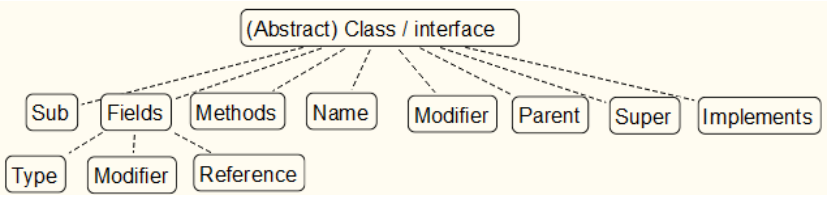

Figure 7.5: Interfaces, abstract classes, and classes in the pseudo-AST representation

METHODS

A method definition of "normal" methods can be seen in figure 7.6. A method has a signature, can have fields, can have a return type, it has a body, a modifier and it has a parent subject. The pseudo-AST representational subjects for methods are:

- Signature - A signature is an abstraction that can be used to differentiate between methods. It consists out of the name, the parameters, and the parameter types of a method. This leaf is Default.
- Fields - A list of all fields in that method. A field can be a variable declaration. It can also be a reference to an instance of a class. The field can be public, private, or any other modifier just like in the definition of the programming language Java.
- Parent - A reference to the parent of that method. Which can be a class, an abstract class, or an interface.
- Body - Contains a reference to the body of this method.

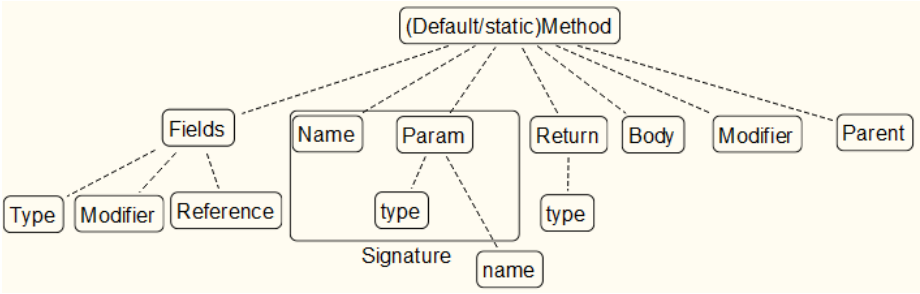

Figure 7.6: Methods in the pseudo-AST representation

Besides default, static or normal methods, there are abstract methods that cannot own fields nor have a body. This can be seen in figure 7.7.

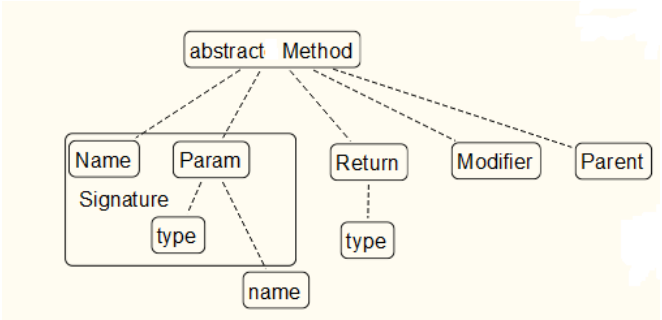

Figure 7.7: Abstract methods in the pseudo-AST representation

METHOD CALLERS

A caller does a method call. It has parameters, a reference, a parent, and a keyword. This definition can be seen in figure 7.8. The pseudo-AST representational subjects for method callers are:

- Params - A list with the parameters which are transferred to the method.
- Reference - A reference to the method that is to be called. This leaf is Default.
- Parent - A reference to the parent of that method. Which can be a class or an interface.
- Keyword - Like super. Where a method call refers to the method in a superclass

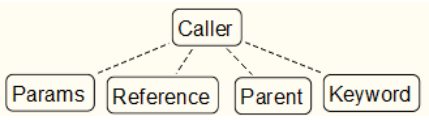

Figure 7.8: Callers in the pseudo-AST representation

FIELDS

A field is a variable declaration of a primitive type like int or a non-primitive type like an object reference. A field has a name, a modifier, and in the case of a non-primitive type a reference to an object or instance. This definition can be seen in figure 7.9. The pseudo-AST representational subjects for fields are:

- Name - The name of the field.
- Modifier - Keywords final, static, and others apply.
- Parent - A reference to the parent of that field. Which can be a class, an interface, or a method.
- Instance - A reference to the instance of a field. Which is an object reference.
- Type - Primitive like int, or non-primitive like an object type (class)

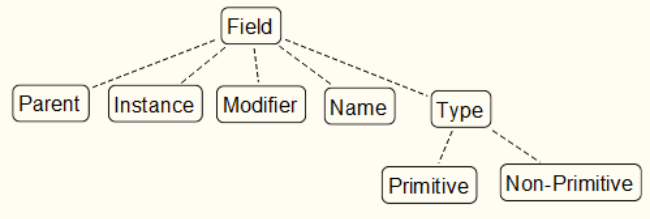

Figure 7.9: Fields in the pseudo-AST representation

INSTANTIATION

An instantiation is the creation of a new instance of an object (ie. class). An instantiation is known for its keyword "new". An instantiation has a receiver, a parent, parameters, and if there is a constructor, the constructor is referenced. The parameters concern the parameters for the constructor. This definition can be seen in figure 7.10. The pseudo-AST representational subjects for instantiation are:

- Receiver - The receiver is the receiving object or class.
- Parent - A reference to the parent of that method. Which can be a class, an interface, or a method.
- Param - A reference to the parameters of the constructor. If used.
- Constructor- A reference to the constructor of a class.

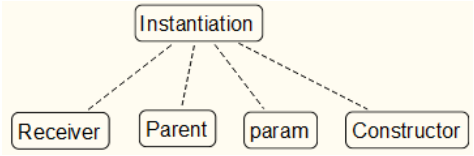

Figure 7.10: Instantiations in the pseudo-AST representation

FIELDASSIGNMENT

A fieldassignment is where values or object references get assigned to a field or variable. For example, the definition of the field test, which is primitive type int. A fieldassignment occurs where a value or an object reference gets assigned to a field. A fieldassignment has a name, which is the name of the referenced field. It has a receiver. It has a parent, which is its parent subject. And it has a type, which is the type of the field where it is referenced to. The type can be an instance of a class as well. This definition can be seen in figure 7.11. The pseudo-AST representational subjects for fieldassignments are:

- Name - The name of the referencing field.
- Receiver - The receiver is a reference to the receiving field.
- Parent - A reference to the parent of that method. Which can be a class, an interface, or a method.
- Type - References the type of the receiving object. Which can be a primitive type or non-primitive (object reference).

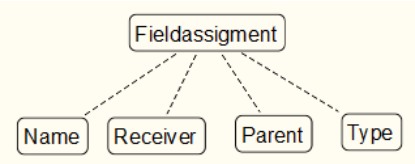

Figure 7.11: Fieldassignment in the pseudo-AST representation

Field Access

A fieldaccess is the usage of a field or variable. For example, the definition of the field test, which is primitive type int. A fieldaccess is where this field "test" gets used. A fieldaccess has a name, which is the name of the referenced field. It has a receiver which is the receiving subject. It has a parent, which is its parent subject. And it has a type, which is the type of the field where it is referenced to. This definition can be seen in figure 7.12. The pseudo-AST representational subjects for fieldaccess are:

- Name - The name of the referencing field.
- Receiver - The receiver is a reference to the receiving field.
- Parent - A reference to the parent of that method. Which can be a class, an interface, or a method.
- Type - References the type of the receiving object. Which can be a primitive type or non-primitive (object reference).

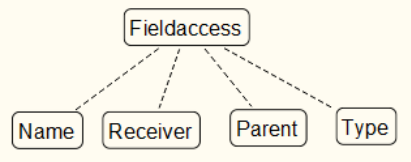

Figure 7.12: Fieldaccess in the pseudo-AST representation

Scopable elements in the new pseudo-AST representation

Some subjects in de pseudo-AST representation are changed to make the query language easier to work with. To ensure that the correct predefined scopable elements are used in the query language, a reference to the scopable elements is presented in figure 7.13. Although this reference for scopable elements is provided as a tree in a hierarchical order, it should just be interpreted as a pick-list reference. If an element is not in the tree in figure 7.13, it cannot be used as a scopable element in the query language. A remark is that the list of scopable elements in figure 7.13 can be changed to adapt to future changes. The pseudo-AST representational scopable elements can be seen in figure 7.13 below.

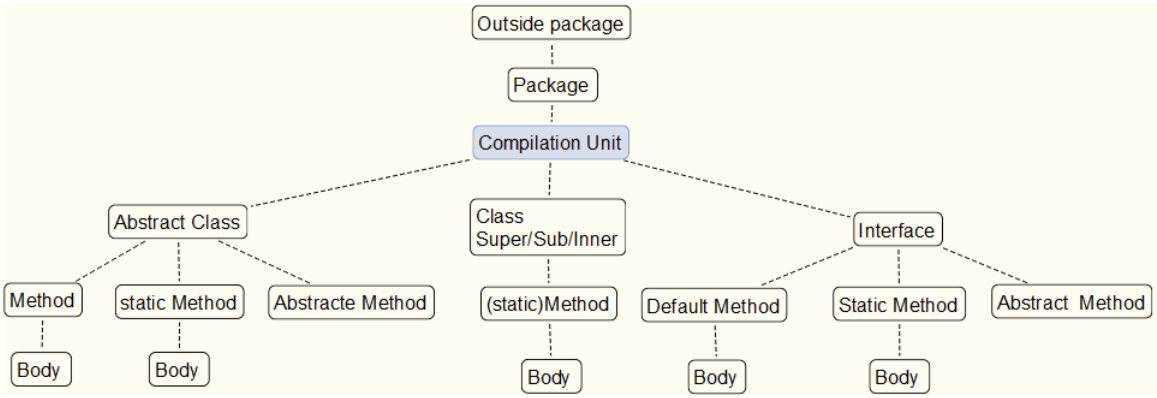

Figure 7.13: scopable elements in the pseudo-AST representation

7.1.5. THE NOTATION OF SCOPABLE ELEMENTS, SUBJECTS, AND FUNCTIONS IN A STATEMENT IN THE QUERY LANGUAGE

The language that defines the query language will be defined in this section. This language will be used to prove the research questions raised in this study. The examples below are based on a hypothetical situation, where the pseudo-AST representation in figure 7.14 is derived from a source code entity, which is presented in figure 7.15:

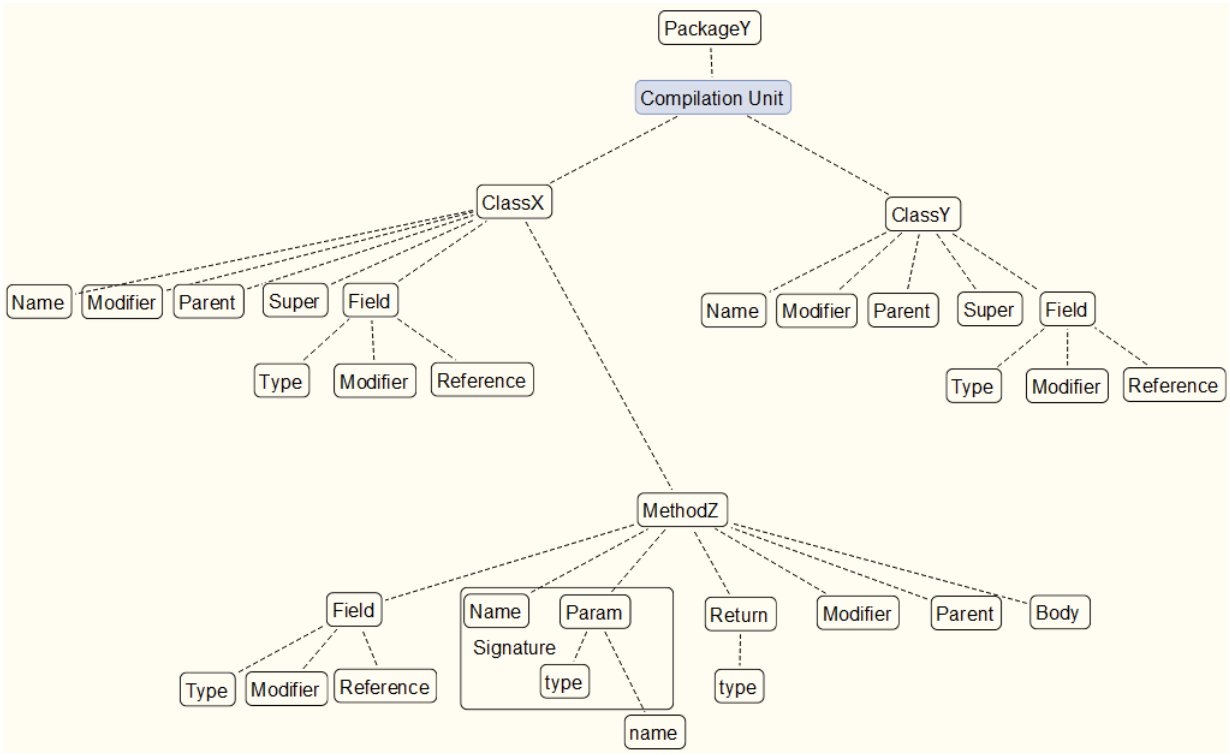

Figure 7.14: The hypothetical pseudo-AST representation example derived from the code example in figure 7.15

```
package Y;

public class X
	{
	public int Z(int paramX)
		{
		return(paramX);
}
	}

public class Y
	{
	public int classAttribute=0;
}
```

Figure 7.15: The hypothetical code example to be used in this chapter.

In this hypothetical situation, there is a PackageY, a ClassX, a ClassY, a methodZ, and an attribute with the name classAttribute.

NOTATION OF SCOPABLE ELEMENTS

Some examples will be presented to make the usage of the notation of scopable elements in query language statements clear. The notation of a scopable element is defined as:

$$\textbf{@ scope} \tag{7.1}$$

The hypothetical situation from figure 7.14 is used for the examples below.

ex.1: **@ClassX** , which means in class X.
ex.2: **@PackageY** , which means in package Y.
ex.3: **@ALL** , which is an abstraction of everything in the pseudo-AST.

THE NOTATION OF SUBJECTS COMBINED WITH SCOPABLE ELEMENTS

Subjects are always accompanied by a scopable element. To define a subject, it must be referred to its scope. The definition of the subject notation:

$$\textbf{subject}\ @scope \tag{7.2}$$

The examples below use the scope to specify the exact subject in the pseudo-AST representation. The hypothetical situation from figure 7.14 is used for the examples below.

ex.4: **ClassY**@PackageY , which means in class X in (scope of) package Y.
ex.5: **MethodZ**@ClassX , which means method Z in class X.

Examples 6 to 8 refer to a subject's leaf or end-node. Their end-nodes or leaves can be seen in figure 7.5. The hypothetical situation from figure 7.14 is used for the examples below.

ex.6: **Parent**@ClassX , which means the parent subject of class X. Which is PackageY.
ex.7: **Super**@ClassX , which means the superclass of class X. Which yields an empty list.
ex.8: **Implements**@ClassX , The list of interfaces that ClassX implements. Which is none.
ex.9: **Modifier**@ClassX , The list of modifiers of ClassX. Which is "Public" in this example.

Definition 7.1.6 is applied which leads to a pipelined example.

ex.10: **Modifier**@MethodZ@ClassX , which is the modifier of Method Z in class X. Which is Public.

Although pipelining makes cascading of statements possible, the usage of long cascaded statements should be prevented.

SHORTENING NOTATIONS WITH ALIASES

Aliases can improve the readability and the reuse of statements. An alias can refer to every statement or list. An alias should be printed in bold to differentiate between subjects and scopable elements. Further, the alias its name should represent its content or meaning. An alias is defined by using a colon with a single space at the end of a statement. Empty aliases are not allowed and its name must be unique.

$$statement: \textbf{alias} \tag{7.3}$$

ex.11: Modifier @ ClassX: **TheModifierOfClassX** , Alias TheModifierOfClassX now refers to the modifier(s) of class X.

ex.12: MethodZ @ ClassX: **MethodZOfClassX** , alias MethodZOfClassX now refers to the subject MethodZ in the pseudo-AST representation.

ex.13: Modifier @ **MethodZOfClassX** , Pipelined and reused alias to represent example ex.10 in two stages with a new alias.

ex.14: param @ **MethodZOfClassX**: **ParamOfMethodZInClassX** , the param subject of the method Z in Class X with a new alias.

ex.15: type @ **ParamOfMethodZInClassX** , the type subject of the method Z in Class X.

ex.16: signature@**MethodZOfClassX**: **SignatureOfMethodZ** , the type subject of the method Z in Class X.

Statements can be build-up in stages by using aliases, which increases readability. Also, query language statements that are often used can be aliased to increase their reuse.

ACTIONS TO BE PERFORMED ON OUTPUT

Since the output from a statement is always a list with no, one or multiple entities in an AST representation, there is a need for a filter function that has not yet been defined. A filter has an input which is a list and it needs an expression. The expression results in a TRUE or FALSE for each element in the input list. When the expression is TRUE, the element for which the expression is TRUE will be added to the output list. When FALSE, the value will not be added to the output list. A filter action will always start with the word FILTER written in capital letters. The output of the filter is a list with no-, one-, or multiple elements.

$$\textbf{FILTER [expression](list)} \tag{7.4}$$

ex.16: FILTER [modifier=private](**MethodZOfClassX**) , which filters the list of modifiers from Method Z of Class X and outputs either nothing or exactly one method that has the keyword that matches the expression.

ASSIGNMENTS TO ALIASES

To increase the expressibility of the query language, assignments, and modifications to aliases should be possible. An example of an assignment/modification is when a new signature has to be defined to detect its presence in the pseudo-AST representation. Assume that there is a signature of method Q. We need to add a parameter to this method Q of the type int and we need to detect that this new signature of method Q is not yet present. If it is, there will be an error that the method is already defined. An alias can be assigned to an existing alias (which represents a list) and optionally subjects can be added or removed from that existing alias and stored into a new alias. The alias labeled "existing alias" on the right part of equation 7.5 will remain unaltered.

$$\textbf{new alias = existing alias ++/– [optional subject]} \tag{7.5}$$

ex.17: **MyAlteredSignature**=**SignatureOfMethodZ** ++ [Param(TypeY,NameY)] , supplies the old signature of Method Z with an extra parameter of name Y and Type Y and assigns it to a new Alias. This assignment has to be repeated to add more than one parameter.

ex.18: **MyAlteredSignature**=**SignatureOfMethodZ** – [Param(TypeY,NameY)] , removes a extra parameter of name Y and Type Y from the signature of method Z and assigns it to a new alias.

ex.19: **MyCopiedSignature**=**SignatureOfMethodZ** , assigns the signature of method Z to a new alias.

OPERATIONS ON OUTPUTS OF STATEMENTS

The query language will provide for operations on outputs (lists) of statements, like unions and intersections. The output of such an operation is a list.

$$\textbf{ListA [union/intersects]ListB} \tag{7.6}$$

ex.20: **MyTestParameter**= ++ [Param(TypeY,NameY)] , creates a new alias **MyTestParameter** (if it does not exist) with a parameter of type Y and with name Y. Since this alias is a new and empty list, a parameter is added to that empty list. If the alias already existed and the '++' symbols had been left away, the alias was overwritten by the new parameter with NameY of TypeY.

ex.21: **ParamOfMethodZInClass**[intersects]**MyTestParameter** , intersects the list of the alias MyTestParameter with the list of all Parameters in class Z. This is similar to a filter function. The filter function has similar behavior to the intersection operation.

7.1.6. DETECTORS

Detectors detect subjects within a scope. A detector needs a subject type and a scope. The output of a detector is a list of detected subjects. Detectors can be aliased and therefore reused. The definition of a detector is determined by the question mark at the end of the subject type:

subject?@scope: ALIAS (7.7)

ex.22: method?@**ClassX** , detects all method subjects within the scope of ClassX. Which is just one, MethodZ.

ex.23: class?@ALL: **ClassInAll** , detects all classes in ALL, which are in this case, two classes, ClassX and ClassY. Also, an alias ClassInAll is created.

ex.24: method?@**ClassInAll: MethodInAll** , detects all methods in the two classes ClassX and ClassY, which is in this case, one method, MethodZ, and creates an alias MethodInAll.

ex.25: signature?@**MethodInAll: AllSignaturesInAll** , detects all signatures present in ALL and creates a reusable alias for it.

ex.25: signature?@MethodZ: **SignatureOfMethodZ** , detects/retrieves one specific signature of methodZ. Simple actions like retrieving a signature from a known method can also be performed.

ex.26: caller?@MethodZ@ALL: **AllCallersInMethodZ** , detects all method-callers of MethodZ in All.

ex.27: caller?@MethodZ@ClassX: **AllCallersInMethodZ** , detects all method-callers of MethodZ which are in the scope of ClassX.

This last definition of the detectors concludes the query language to be used as a tool.

7.2. MICROSTEP INDUCED HAZARD

Making changes to a source code is a hazardous activity, but it is not clear what that change implies and what harm is invoked with that hazardous activity. The mechanic refactoring steps from Fowler are reduced to microsteps, but it is not clear if Table 5.2 with microsteps and their subject can cover the majority of Fowler's refactorings. If the table with microsteps is considered complete, the hazards per microsteps should be specified. Finally, the list of hazards per microstep should be complemented with hazardous code pattern descriptions.

7.2.1. TRANSFORMING FOWLER'S REFACTORING MECHANIC STEPS INTO MICROSTEP

Fowlers refactoring mechanics will be used to transform each step into a list of microsteps. All steps from the refactoring mechanics are collected and transformed into microsteps. Since microsteps do not include checks, and/or compile-and-test, these can be removed from the mechanics. The transformation of Fowler's mechanic steps to microsteps is presented below in table 7.1 and table 7.2. A cascade of microsteps means that these microsteps are executed in sequential order.

Type	Fowler's refactoring steps	Microstep
Method	Declare a new method with signature and body	Add method
	Remove old method	Remove method
	Rename method	Remove method + Add method cascade
	Replace the body of a method with the body of the old method	Remove method + Add method cascade
	Add parameter to method	Remove method + Add method cascade
	Remove a parameter to method	Remove method + Add method cascade
	Change modifier for method	Remove method + Add method cascade
	Remove modifier for method	Remove method + Add method cascade
	Add modifier to method	Remove method + Add method cascade
	Change method return type	Remove method + Add method cascade
	Create a field	Add field
	Remove a field	Remove field
	Rename a field	Remove field + Add field cascade
	Create getter for a field	Add method
	Create setter for a field	Add method
	Change modifier for field/attribute	Remove field + Add field cascade
	Remove modifier for field/attribute	Remove field + Add field cascade
	Add modifier to field/attribute	Remove field + Add field cascade
	Change the field type	Remove field + Add field cascade

Table 7.1: Fowler's refactoring mechanic steps to microsteps table part 1

Part 2 of the table will continue on the next page.

Type	Fowler's refactoring steps	Microstep
Class	Create a new class	Add class
	Remove a class	Remove Class
	Rename a class	Remove Class + Add Class cascade
	Move a class with a new name and existing body	Remove Class + Add Class cascade
	Change modifier for class	Remove Class + Add Class cascade
	Remove modifier for class	Remove Class + Add Class cascade
	Add modifier to class	Remove Class + Add Class cascade
Interface	Rename interface	Remove interface + Add interface cascade
	remove interface	Remove interface
	add interface	Add interface
	Change modifier for interface	Remove interface + Add interface cascade
	Remove modifier for interface	Remove interface + Add interface cascade

Table 7.2: Fowler's refactoring mechanic steps to microsteps table part 2

This list of microsteps covers a selection of Fowler's mechanic refactoring steps. This selection is made from as many refactorings as needed to cover all microsteps.

7.2.2. What will the hazard(s) of a particular microstep be?

In principle all microsteps are hazardous. Removing subjects, with the remove method microstep, can induce harm because callers to that method are no longer valid. The same principle is valid for adding subjects, like adding an attribute. A similar attribute with the same name can already be present. The hazards per microsteps will be discussed in this section. The hazard(s) that can occur with microstep activity will be presented for each microstep. The short label (Short) of the hazards represents the microstep and the hazard id.

Microsteps that concern Class subjects

Microstep **Add Class** adds a class in a source code entity. A class has a name and a body. See table 7.3 below for the hazard definitions of the Add Class microstep:

Microstep	Class	Attribute	Method	Interface	Parameter
Add	X				
Remove					
Rename					

Short	Hazard Add Class	Type	Severity
AC-H1	Class is already defined	Compile error	Low

Table 7.3: Microsteps Add class with hazard(s)

Microstep **Remove Class** removes a class in a source code entity. See table 7.4 below:

Microstep	Class	Attribute	Method	Interface	Parameter
Add					
Remove	X				
Rename					

Short	Hazard Remove Class	Type	Severity
RC-H1	Class is still in use.	Compile error	Low
RC-H2	Class is removed from the inheritance tree.	Compile error	Low
RC-H3	Static method from Inner static class still referenced from ALL, and the outerclass is going to be removed.	Compile error	Low
RC-H4	Class is removed and possible design pattern is broken, due to abstract class definition.	Compile error	Low

Table 7.4: Microsteps Remove Class with hazard(s)

Table 7.4 for **Remove Class** will be redefined in Table 7.5 since the hazards RC-H1 and RC-H2 consists out of multiple separate hazards. ie. in use can comprehend that a method or an attribute of that class can still be referenced.

Short	Hazard Remove Class	Type	Severity
RC-H1-1	Instantiations (keyword new) of a class will become invalid.	Compile error	Low
RC-H1-2	Object reference to class will become invalid. This includes method calls or attributes	Compile error	Low
RC-H1-3	Caller to a method in a class is still present in the source tree.	Compile error	Low
RC-H2-1	Subclass is removed from the inheritance tree.	Compile error	Low
RC-H2-2	Superclass is removed from the inheritance tree.	Compile error	Low
RC-H3	Static method from Inner static class still referenced from ALL, and the outerclass is going to be removed.	Compile error	Low
RC-H4	Class is removed and possible design pattern is broken, due to abstract class definition.	Compile error	Low

Table 7.5: Remove Class microsteps with their hazard(s)

Microstep **Rename Class** renames a class in a source code entity. See table 7.6 below.

Microstep	Class	Attribute	Method	Interface	Parameter
Add					
Remove					
Rename	X				

Table 7.6: Microstep rename class

The **Rename Class** microstep is identical to a cascaded action of the add class microstep with a new name and the old body and the remove class microstep with its old name and body. See figure 7.16. Important notice about the rename microstep is that the hazards of the separate remove- and add Class microstep cascade as well.

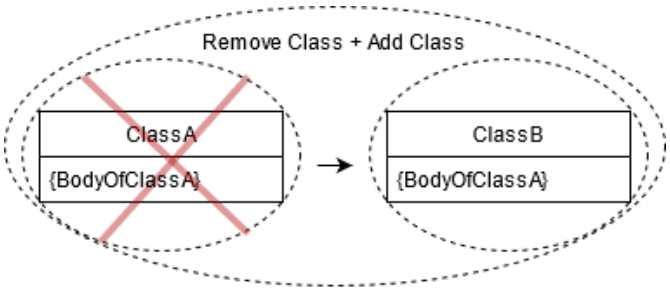

Figure 7.16: Rename Class as a cascaded action of a remove- and add Class microstep

MICROSTEPS THAT CONCERN ATTRIBUTE SUBJECTS

Microstep **Add Attribute** adds an attribute to a class in a source code entity. An attribute has a name, a type, and a modifier. See table 7.7 below for the hazards of the Add attribute microstep.

Microstep	Class	Attribute	Method	Interface	Parameter
Add		X			
Remove					
Rename					

Short	Hazards Add Attribute	Type	Severity
AA-H1	Attribute already defined	Compile error	Low
AA-H2	Inner-/outer class attribute redefinition (Attribute from outer class is used in inner class and new attribute with same name and type is added to inner class)	Program Preservation Break	Major
AA-H3	Inheritance redefined attribute in Subclass (Attribute from superclass is used in a sub class and new attribute with same name and type is added to that sub class)	Program Preservation Break	Major

Table 7.7: Microsteps Add Attribute with hazard(s)

Microstep **Remove Attribute** Removes an attribute from a class in a source code entity. An attribute has a name, a type, and a modifier. See table 7.8 below for the hazards of the Remove attribute microstep"

Microstep	Class	Attribute	Method	Interface	Parameter
Add					
Remove		X			
Rename					

Short	Hazards Remove Attribute	Type	Severity
RA-H1	Attribute still in use.	Compile error	Low
RA-H2	Inner-/outer class attribute redefinition (Attribute from outer class is used and declared in inner and outer class and attribute is removed from inner class)	Program Preservation Break	Major
RA-H3	Inheritance redefined attribute in Subclass (Attribute from superclass is used and declared in super- and in subclass and subclass attribute is removed).	Program Preservation Break	Major

Table 7.8: Microsteps Remove Attribute with hazard(s)

Microstep **Rename Attribute**, renames an attribute in a class in a source code entity. See table 7.9 below:

Microstep	Class	Attribute	Method	Interface	Parameter
Add					
Remove					
Rename		X			

Table 7.9: Microstep rename attribute

The rename attribute microstep is identical to a cascaded action of the add attribute microstep and the remove attribute microstep. An important notice is that when the microsteps are cascaded, the hazards of the separate microsteps cascade as well.

MICROSTEPS THAT CONCERN METHOD SUBJECTS

Microstep **Add Method** adds a method to a source code entity. A method has a name, a type, a modifier, a signature, and a body. See table 7.10 below for the hazards of the Add method microstep.

Microstep	Class	Attribute	Method	Interface	Parameter
Add			X		
Remove					
Rename					

Short	Hazards Add Method	Type	Severity
AM-H1	Method already defined	Compile error	Low
AM-H2	Inheritance Redefined Method in Subclass. caller from subclass calls Subclass method instead of superclass method, after adding a method to the subclass.	Program Preservation Break	Major

Table 7.10: Microsteps Add method with hazard(s)

Microstep **Remove Method** Removes a method from in a source code entity. A method has a name, a type, a modifier, a signature, and a body. See table 7.11 below for the hazards of the Remove Method microstep.

Microstep	Class	Attribute	Method	Interface	Parameter
Add					
Remove			X		
Rename					

Short	Hazards Remove Method	Type	Severity
RM-H1	Callers to method are incorrect.	Compile error	Low
RM-H2	Inheritance Redefined Method in Subclass. caller from subclass calls Superclass method instead of subclass method, after removing a method from the subclass.	Program Preservation Break	Major
RM-H3	Method definition in an interface.	Compile error	Low

Table 7.11: Microsteps Remove method with hazard(s)

Microstep **Rename Method** renames a method in a source code entity. A method has a name, a type, a modifier, and a signature. See table 7.12 below:

Microstep	Class	Attribute	Method	Interface	Parameter
Add					
Remove					
Rename			X		

Table 7.12: Microstep rename method

The rename method microstep is identical to a cascaded action of the add method microstep and the remove method microstep. An important notice is that when the microsteps are cascaded, the hazards of the separate microsteps cascade as well.

MICROSTEPS THAT CONCERN INTERFACE SUBJECTS

Microstep **Add Interface** adds an interface to a source code entity. An interface has a name, a modifier, and a body. See table 7.13 below for the hazards of the Add Interface microstep.

Microstep	Class	Attribute	Method	Interface	Parameter
Add				X	
Remove					
Rename					

Short	Hazards Add Interface	Type	Severity
AI-H1	Interface already defined.	Compile error	Low

Table 7.13: Microsteps Add Interface with hazard(s)

Microstep **Remove Interface** removes an interface from a source code entity. An interface has a name, a modifier, and a body. See table 7.14 below for the hazards of the Remove Interface microstep.

Microstep	Class	Attribute	Method	Interface	Parameter
Add					
Remove				X	
Rename					

Short	Hazards Remove Interface	Type	Severity
RI-H1	Interface still in use by a class.	Compile error	Low
RI-H2	Interface still in use by another interface.	Compile error	Low
RI-H3	Interface holds static/default implementations.	Compile error	Low

Table 7.14: Microsteps Remove interface with hazard(s)

Microstep **Rename Interface** renames an interface in a source code entity. An interface has a name, a modifier, and a body. See table 7.15 below:

Microstep	Class	Attribute	Method	Interface	Parameter
Add					
Remove					
Rename				X	

Table 7.15: Microstep rename interface

The rename interface microstep is identical to a cascaded action of the add interface microstep and the remove interface microstep. An important notice is that when the microsteps are cascaded, the hazards of the separate microsteps cascade as well.

MICROSTEPS THAT CONCERN PARAMETER SUBJECTS

Microstep **Add Parameter** adds a Parameter to a source code entity (ie. a method). A parameter has a name and a type. See table 7.16 below for the hazards of the Add Parameter microstep.

Microstep	Class	Attribute	Method	Interface	Parameter
Add					X
Remove					
Rename					

Short	Hazards Add Interface	Type	Severity
AP-H1	callers to method became incorrect after adding a parameter.	Compile Error	Low
AP-H2	Method already defined after adding a parameter.	Compile Error	Low
AP-H3	caller from subclass calls Subclass method instead of superclass method, after adding a parameter to the method in the subclass.	Program Preservation Break	Major
AP-H4	Method definition in an interface. Interface contract broken after adding a parameter.	Compile error	Low
AP-H5	Parameter of method with the same name is already present	Compile error	Low
AP-H6	Field with the same name is already in use in Method	Compile error	Low

Table 7.16: Microsteps Add Parameter with hazard(s)

Microstep **Remove Parameter** removes a parameter in a source code entity. A parameter has a name and a type. See table 7.17 below for the hazards of the Remove Interface microstep.

Microstep	Class	Attribute	Method	Interface	Parameter
Add					
Remove					X
Rename					

Short	Hazards Remove Interface	Type	Severity
RP-H1	callers to method became incorrect after removing a parameter.	Compile Error	Low
RP-H2	Method already defined after removing a parameter.	Compile Error	Low
RP-H3	caller from subclass calls Subclass method instead of superclass method, after removing a parameter to the method in the subclass.	Program Preservation Break	Major
RP-H4	Method definition in interface. Interface contract broken after removing a parameter.	Compile error	Low
RP-H5	Parameter still in use in Method.	Compile error	Low

Table 7.17: Microsteps Remove Parameter with hazard(s)

Microstep **Rename Parameter** renames a parameter in a source code entity. A parameter has a name and a type. See table 7.18 below:

Microstep	Class	Attribute	Method	Interface	Parameter
Add					
Remove					
Rename					X

Table 7.18: Microstep Rename Parameter

The **Rename Parameter** microstep is identical to a cascaded action of the add parameter microstep and the remove parameter microstep. An important notice is that when the microsteps are cascaded, the hazards of the separate microsteps cascade as well.

7.2.3. WHICH HAZARDOUS CODE PATTERN DESCRIPTIONS ARE NEEDED FOR EACH MICROSTEP'S INDUCED HAZARD TO DETECT POTENTIAL SOFTWARE HARM?

The previous section defined the hazards per microstep. This section will focus on what hazardous code pattern descriptions accompanies a microstep's induced hazard to detect potential software harm. The hazardous code pattern descriptions will be described with the query language per defined microstep induced hazard. A hazardous code pattern description will be created for the hazard AC-H1 in table 7.19 to demonstrate the power of the query language. This Table is a copy of Table 7.3 to enhance the reading convenience.

Short	Hazard Add Class	Type	Severity
AC-H1	Class is already defined	Compile error	Low

Table 7.19: Microsteps Add class with hazard(s)

This hazard describes that the hazardous code pattern descriptions for this hazard exist out of the presence of a class definition with the same name within the source code entity at the package level. Important notice for this definition is that the name of the new to-be-added class should be known. To detect potential software harm, the harmful code fragment needs to be detected in the package. The scopable elements figure from 7.17 is used to select the right subjects. The scope is defined at package-level, since a name col-

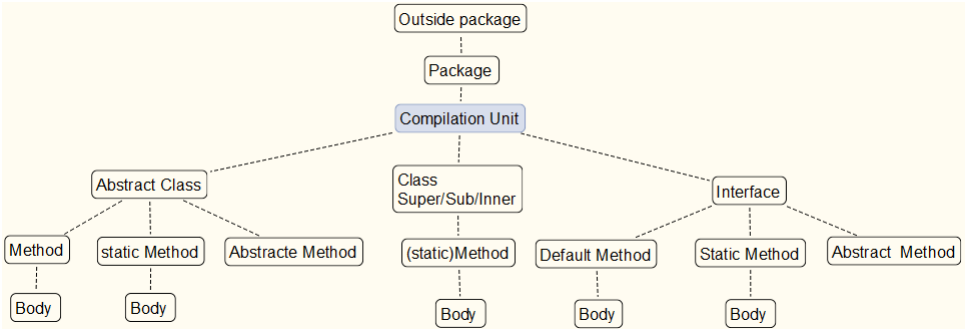

Figure 7.17: scopable elements in the pseudo-AST representation

lision on class level can only occur within the scope of a package. The subject is the class that is to be added, from which the name is yet known. This class will be aliased as the **iNewclass**. Since the presence of this harmful code fragment needs to be detected, the statement will be declared as a detector with a question mark. Since the end-node or leaf from the class subject is the name of the class, there is no need to express the end-node "name" of the subject class. ie. name@class. The final query language statement for (AC-H1) is: **iNewclass**?@package. Where the **iNewclass** alias refers to the name of the to-be-added class. This exact statement is the hazardous code pattern description that detects the hazardous code fragments in the source code entity for the hazard AC-H1.

Another example is a relatively large and complex harmful code pattern description for the microstep Add Method with its induced Hazard H2, which is part of the harmful trigger event AM-H2-HCPD. Step by step the statements of the query language which represents this harmful code pattern description will be explained.

A problem of the Add Method microstep arises when there is a superclass with a subclass. The subclass has a method call, that does not use the keyword super, which calls a method in the superclass. Now, a method will be added to the subclass, assume this is a method with a signature represented by the alias **iNewSignature**. The signature of this new method and the class where this new method will be added must be known. If the signature represented by the alias **iNewSignature** of the newly added method in the subclass is the same as the signature of the Method from the superclass, program preservation break occurs because the caller from the subclass now calls the subclass method instead of the superclass method. The harmful code pattern description for the add method microstep hazard #2 (AM-H2-HCPD) is explained below:

super?@**iClassOfMethod2Add**: **ClassOfMethodHasSuperAM**
Checks if the class iClassOfMethod2Add has a superclass and creates an alias for it.

sub?@**iClassOfMethod2Add**: **ClassOfMethodHasSubAM**
Checks if the class iClassOfMethod2Add has a subclass and creates an alias for it.

FILTER[signature=**iNewSignature**](method?@**ClassOfMethodHasSuperAM**): **MethodIsAtSuperAM**
Checks if the Signature iNewSignature is already defined in the superclass ClassOfMethodHasSuperAM. From all methods in the superclass, find a method with Signature=iNewSignature.

caller?@**MethodIsAtSuperAM**: **iNewMethodToSuperCallAM**
Check if there is a caller to that method in the superclass

FILTER[keyword!='super'](**iNewMethodToSuperCallAM**): **MethodSupCallNoKeyAM**
From that list of callers to the method in the superclass, filter the callers that do not use the keyword 'super'.

MethodSupCallNoKeyAM [intersects]caller?@**ClassOfMethodHasSubAM**: **CallSubToSiginSupNoKeyAM**
Intersect that list of non-super Callers with all callers from the subclass. There is a list of caller(s) to MethodX without the keyword super and we test if there is one in the subclass by intersecting both lists. The final output is the alias **CallSubToSiginSupNoKeyAM**. If empty, the hazardous code fragment is not detected. If one or multiple elements are present, program preservation break danger will happen.

HAZARDOUS CODE PATTERN DESCRIPTIONS THAT CONCERN CLASS MICROSTEPS

The representation of the Hazardous code pattern description for **Add Class** microstep **Hazards** (AC-H1) can be seen in Table 7.20.

input(AC-H1-HCPD): **iNewClass** = input(i) Class to be added.

Harmful Trigger Event	Hazardous code pattern description
AC-H1-HCPD	**newClass**?@package: **ClassAlreadyInPackage**

Table 7.20: Microsteps Add class with Hazardous code pattern descriptions

The representation of the Hazardous code pattern description for **Remove Class** microstep **Hazards** (RC-H) can be seen in Table 7.21.

input(All detectors): **iClass2BRemoved** = The class to be removed.

Harmful Trigger Event	Hazardous code pattern description
RC-H1-1-HCPD	FILTER[receiver=**iClassRemoved**](Instantiation?@**ALL**): **findFieldInstantiationToClass**
RC-H1-2-HCPD	FILTER[instance=**iClass2BRemoved**](field?@**ALL**): **FindFieldReferenceToClass**
RC-H1-3-HCPD	caller?@**iClass2BRemoved@ALL**: **AllCallersToMyClass**
RC-H2-1-HCPD	super?@**iClass2BRemoved**: **IfEmptyNoSuperclassInTree**
RC-H2-2-HCPD	sub?@**iClass2BRemoved**: **IfEmptyNoSubclassInTree**
RC-H3-HCPD	class?@**iClass2BRemoved**: **AreThereClassesInsideAClass** FILTER[modifier='static'](**AreThereClassesInsideAClass**): **StaticInnerClasses** method?@**StaticInnerClasses**: **MethodsOfStaticInnerClasses** FILTER[modifier='static'](**MethodsOfStaticInnerClasses**): **StaticMethodsInStaticInnerClasses** caller?@**StaticMethodsInStaticInnerClasses@ALL**: **ListOfCallersToStaticMethodInStaticInnerClass**
RC-H4-HCPD	FILTER[modifier='astract'](**iClass2BRemoved**): **FindClassAbstractModifier**

Table 7.21: Microsteps Remove Class hazards with Hazardous code pattern descriptions

For **Rename Class** microstep a cascade of the add and remove class hazardous code fragments have to be detected.

HAZARDOUS CODE PATTERN DESCRIPTIONS THAT CONCERN ATTRIBUTE MICROSTEPS

The representation of the Hazardous code pattern description for **Add Attribute** microstep **Hazards** (AA-H) can be seen in Table 7.22.

input(All AA detectors): **iIsAttributeToAdd** = The attribute that will be added.
input(All AA detectors): **iClassOfAttributeToAdd** = The class that the new attribute will be add to.

Harmful Trigger Event	Hazardous code pattern description
AA-H1-HCPD	FILTER[name=**iIsAttributeToAdd**](field?@**iClassOfAttributeToAdd**): **FindFieldWithNameInClass**
AA-H2-HCPD	parent?@**iClassOfAttributeToAdd**: **ParentOfClassThatHasAttribute** **ParentOfClassThatHasAttribute**[intersects]class?@package: **ParentThatHasAttributeIsClass** FILTER[name=**iIsAttributeToAdd**](field?@**ParentThatHasAttributeIsClass**): **ParHasAttrAA** FILTER[name=**ParHasAttrAA**](Fieldassignment?@**iClassOfAttribute**): **ListAssignmentsOfFieldsInClassAA**
AA-H3-HCPD	super?@**iClassOfAttributeToAdd**: **SuperClassOfClassThatHasAttribute** FILTER[name=**iIsAttributeToAdd**](field?@**SuperClassOfClassThatHasAttribute**): **SupHasAttrAA** FILTER[name=**SupHasAttrAA**](Fieldassignment?@**iIsAttributeToAdd**): **AssignFieldsInSupOfClassAA**

Table 7.22: Microsteps Add Attribute hazards with Hazardous code pattern descriptions

The representation of the Hazardous code pattern description for **Remove Attribute** microstep **Hazards** (RA-H) can be seen in Table 7.23.

input(All RA detectors): **iAttr2Remove** = The attribute to be removed

Harmful Trigger Event	Hazardous code pattern description
RA-H1-HCPD	class?@**iAttr2Remove**: **ClassOfAttr2Remove** FILTER[Receiver=**iAttr2Remove**]**(Fieldassignment?@ClassOfAttr2Remove): AssignToOldAttrRef**
RA-H2-HCPD	class?@**iAttr2Remove**: **ClassOfAttr2Remove** parent?@**ClassOfAttr2Remove**: **ParentOfClassThatAttr2Remove** **ParentOfClassThatAttr2Remove**[intersects]class?@package: **ParentIsClassRA** FILTER[name=**iAttr2Remove**](field?@**ParentIsClassRA**): **ParHasAttrRA** FILTER[name=**ParHasAttrRA**](Fieldassignment?@**ClassOfAttr2Remove**): **ListAssignFieldsInClassRA**
RA-H3-HCPD	class?@**iAttr2Remove**: **ClassOfAttr2Remove** super?@**ClassOfAttr2Remove**: **SuperclassOfClassThatHasAttr2Remove** FILTER[name=**iAttr2Remove**](field?@**SuperclassOfClassThatHasAttr2Remove**): **SupHasAttrRA** FILTER[name=**SupHasAttrRA**](Fieldassignment?@**ClassOfAttr2Remove**): **AssignFieldsInSupClassRA**

Table 7.23: Microsteps Remove Attribute hazards with Hazardous code pattern descriptions

For **Rename Attribute** microstep the hazardous code pattern descriptions of a cascade of the add and remove Attribute have to be detected.

HAZARDOUS CODE PATTERN DESCRIPTIONS THAT CONCERN METHOD MICROSTEPS

The representation of the Hazardous code pattern description for **Add Method** microstep **Hazards** (AM-H) can be seen in Table 7.24.

input(All AM detectors): **iNewSignature** = The new method signature of method to be added.
input(All AM detectors): **iClassOfMethod2Add** = The class that new method will be add to.

Harmful Trigger Event	Hazardous code pattern description
AM-H1-HCPD	FILTER[signature=**iNewSignature**](method?@**iClassOfMethod2Add**): **MethodAlreadyDefinedInClassAM**
AM-H2-HCPD	super?@**iClassOfMethod2Add**: **ClassOfMethodHasSuperAM** sub?@**iClassOfMethod2Add**: **ClassOfMethodHasSubAM** FILTER[signature=**iNewSignature**](method?@**ClassOfMethodHasSuperAM**): **MethodIsAtSuperAM** caller?@**MethodIsAtSuperAM**: **iNewMethodToSuperCallAM** FILTER[keyword!='super'](**iNewMethodToSuperCallAM**): **MethodSupCallNoKeyAM** **MethodSupCallNoKeyAM** [intersects]caller?@**ClassOfMethodHasSubAM**: **CallSubToSiginSupNoKeyAM**

Table 7.24: Microsteps Add Method hazards with Hazardous code pattern descriptions

The representation of the Hazardous code pattern description for **Remove Method** microstep **Hazards** (RM-H) can be seen in Table 7.25.

input(All RM detectors): **iOldMethod2Remove** = Method to be removed.

Harmful Trigger Event	Hazardous code pattern description
RM-H1-HCPD	signature?@**iOldMethod2Remove**: **Signature2Remove** FILTER[signature=**Signature2Remove**](caller?@**ALL**): **FindCallersToSignature2BRemoved**
RM-H2-HCPD	signature?@**iOldMethod2Remove**: **Signature2Remove** class?@**iOldMethod2Remove**: **ClassOfMethod2Remove** super?@**ClassOfMethod2Remove**: **ClassOfMethodHasSuperRM** sub?@**ClassOfMethod2Remove**: **ClassOfMethodHasSubRM** FILTER[signature=**Signature2Remove**](method?@**ClassOfMethodHasSuperRM**): **MethodIsAtSuperRM** caller?@**MethodIsAtSuperRM**: **NewMethodToSuperCallRM** FILTER[keyword!='super'](**NewMethodToSuperCallRM**): **MethodSupCallNoKeyRM** **MethodSupCallNoKeyRM** [intersects]caller?@**ClassOfMethodHasSubRM**: **CallSubToSiginSupNoKeyRM**
RM-H3-HCPD	signature?@**iOldMethod2Remove**: **Signature2Remove** package?@**iOldMethod2Remove**: **Method2BremovedPackage** Interface?@**Method2BremovedPackage**: **AllInterfacesInPackageRM** FILTER[signature=**Signature2Remove**](method?@ **AllInterfacesInPackageRM**): **MethodDefInterfRM** FILTER[modifier='static'||modifier='default'](**MethodDefInterfRM**): **MethodStaticDefaultDefInInterfaceRM**

Table 7.25: Microsteps Remove Method hazards with Hazardous code pattern descriptions

For **Rename Method** microstep the hazardous code pattern descriptions of a cascade of the add and remove method has to be detected.

Hazardous code pattern descriptions that concern Interface microsteps

The representation of the Hazardous code pattern description for **Add Interface** microstep **Hazards** (AI-H1) can be seen in Table 7.26.

input: **iInterface2Add** = To be added interface.

Harmful Trigger Event	Hazardous code pattern description
AI-H1-HCPD	**iInterface2Add**?@**ALL**: **NewInterfaceAlreadyDefined**

Table 7.26: Microsteps Add Method hazards with Hazardous code pattern descriptions

The representation of the Hazardous code pattern description for **Remove Interface** microstep **Hazards** (RAI-H) can be seen in Table 7.27.

input(AM-H3-HCPD): **iInterFace2Remove** = Interface to be removed.

Harmful Trigger Event	Hazardous code pattern description
RI-H1-HCPD	FILTER[implements=**iInterFace2Remove**](class?@**ALL**): **ClassThatImplInterfaceToRemove**
RI-H2-HCPD	FILTER[implements=**iInterFace2Remove**](interface?@**ALL**): **InterfaceThatImplInterfaceToRemove**
RI-H3-HCPD	interface?@**ALL**: **AllInterfaces** FILTER[modifier='static'||modifier='default'](method?@**AllInterfaces**): **StaticDefaultImpl** caller?@**StaticDefaultIImpl@ALL**: **StaticInterfaceImplInUse**

Table 7.27: Microsteps Remove Interface hazards with Hazardous code pattern descriptions

For **Rename Interface** microstep the hazardous code pattern descriptions of a cascade of the add and remove Interface have to be detected.

HAZARDOUS CODE PATTERN DESCRIPTIONS THAT CONCERN PARAMETER MICROSTEPS

The representation of the Hazardous code pattern description for **Add Parameter** microstep **Hazards** (AP-H) can be seen in Table 7.28.

input(All AP detectors): **iOldMethod** = Old method of Parameter to add to.
input(All AP detectors): **iTypeNew** = Type of parameter to add
input(All AP detectors): **iNameNew** = Name of parameter to add

Harmful Trigger Event	Hazardous code pattern description
AP-H1-HCPD	signature?@**iOldMethod**: **OldSignature** FILTER[signature=**OldSignature**](caller?@**ALL**): **FindCallersToSignatureWithNewParam**
AP-H2-HCPD	signature?@**iOldMethod**: **OldSignature** class?@**iOldMethod**: **OldMethodClass** **iNewSignature**=**OldSignature** ++ [Param(**iTypeNew**,**iNameNew**)] FILTER[signature=**NewSignature**](method?@**OldMethodClass**): **MethodAlreadyDefinedInClassAP**
AP-H3-HCPD	signature?@**iOldMethod**: **OldSignature** **NewSignature**=**OldSignature** ++ [Param(**iTypeNew**,**iNameNew**)] class?@**iOldMethod**: **OldMethodClass** super?@**OldMethodClass**: **ClassOfMethodHasSuperAP** sub?@**OldMethodClass**: **ClassOfMethodHasSubAP** FILTER[signature=**NewSignature**](method?@**ClassOfMethodHasSuperAP**): **MethodIsAtSuperAP** caller?@**MethodIsAtSuperAP**: **NewMethodToSuperCallAP** FILTER[keyword!='super'](**NewMethodToSuperCallAP**): **MethodSupCallNoKeyAP** **MethodSupCallNoKeyAP** [intersects]caller?@**ClassOfMethodHasSubAP**: **CallSubToSiginSupNoKeyAP**
AP-H4-HCPD	signature?@**iOldMethod**: **OldSignature** package?@**iOldMethod**: **OldMethodpackage** Interface?@**OldMethodpackage**: **AllInterfacesInPackageAP** FILTER[signature=**OldSignature**](method?@ **AllInterfacesInPackageAP**): **MethodDefInInterfaceAP** FILTER[modifier='static'||modifier='default'](**MethodDefInInterfaceAP**): **MethodStatDefDefInInterfAP**
AP-H5-HCPD	param?@**iOldMethod**: **ParamsOfOldMethod** FILTER[name=**iNameNew**](**ParamsOfOldMethod**): **NameIsAlreadyPresentInParamListAP**
AP-H6-HCPD	fields?@**iOldMethod**: **FieldsOfOldMethod** FILTER[name=**iNameNew**](**FieldsOfOldMethod**): **ParamInUseByFieldAP**

Table 7.28: Microsteps Add Parameter hazards with Hazardous code pattern descriptions

The representation of the Hazardous code pattern description for **Remove Parameter** microstep **Hazards** (RP-H) can be seen in Table 7.29.

input(All AP detectors): **iOldMethod** = Old method of Parameter to remove from.
input(All AP detectors): **iTypeRemove** = Type of parameter to Remove
input(All AP detectors): **iNameRemove** = Name of parameter to Remove

Harmful Trigger Event	Hazardous code pattern description
RP-H1-HCPD	signature?@**iOldMethod**: **OldSignature** FILTER[signature=**OldSignature**](caller?@**ALL**): **FindCallersToSignatureWithRemovedParamRP**
RP-H2-HCPD	signature?@**iOldMethod**: **OldSignature** class?@**iOldMethod**: **OldMethodClass** **NewSignature**=**OldSignature** – [Param(**iTypeRemove**,**iNameRemove**)] FILTER[signature=**NewSignature**](method?@**iOldMethodClass**): **MethodAlreadyDefinedInClassRP**
RP-H3-HCPD	signature?@**iOldMethod**: **OldSignature** **NewSignature**=**OldSignature** – [Param(**iTypeRemove**,**iNameRemove**)] class?@**iOldMethod**: **OldMethodClass** super?@**OldMethodClass**: **ClassOfMethodHasSuperRP** sub?@**OldMethodClass**: **ClassOfMethodHasSubRP** FILTER[signature=**NewSignature**](method?@**ClassOfMethodHasSuperRP**): **MethodIsAtSuperRP** caller?@**MethodIsAtSuperRP**: **NewMethodToSuperCallRP** FILTER[keyword!='super'](**NewMethodToSuperCallRP**): **MethodSupCallNoKeyRP** **MethodSupCallNoKeyRP** [intersects]caller?@**ClassOfMethodHasSubRP**: **CallSubToSiginSupNoKeyRP**
RP-H4-HCPD	signature?@**iOldMethod**: **OldSignature** package?@**iOldMethod**: **OldMethodpackage** Interface?@**OldMethodpackage**: **AllInterfacesInPackageRP** FILTER[signature=**OldSignature**](method?@ **AllInterfacesInPackageRP**): **MethodDefInInterfaceRP** FILTER[modifier='static'||modifier='default'](**MethodDefInInterfaceRAP**): **MethodStatDefDefInInterfRP**
RP-H5-HCPD	body?@**iOldMethod**: **BodyOfOldMethod** fieldaccess?@**BodyOfOldMethod**: **AllFieldAccessInsideMethodBody** FILTER[name=**iNameNew**](**AllFieldAccessInsideMethodBody**): **ParamInUseInsideMethodRP**

Table 7.29: Microsteps Remove Parameter hazards with Hazardous code pattern descriptions

For the **Rename Parameter** microstep the hazardous code pattern descriptions of a cascade of the add and remove parameter has to be detected.

7.3. A FLEXIBLE DETECTION MECHANISM TO DETECT FOR POTENTIAL SOFTWARE HARM

The previous chapter elaborated on the method to detect hazardous code fragments which when combined with a microstep, form the harmful trigger event. To detect software harm, the trigger event needs to be detected in the code context and the severity of the accompanying hazard should be known. This chapter will focus on a possible abstract solution that can detect potential software harm by using detectors that detect harmful code fragments. This mechanism will also provide an advice based on the detector- and code context.

7.3.1. HOW CAN HAZARDOUS CODE FRAGMENTS BE DETECTED IN A SOURCE CODE ENTITY FLEXIBLY?

A detector as used in this context detects hazardous code fragments and has an input and an output. The detectors must have the ability to be cascaded or to be put in parallel [1]. Separate detectors must have the ability to be composed into a new detector. A detector must provide a verdict, which is an answer consisting of the entity that it has to detect. The verdict process is an extra layer of abstraction to combine the detector's findings into an answer. When detectors answer atomic questions, this verdict process is least complex, but when the detector is a composition of multiple detectors, the complexity of the verdict process rises.

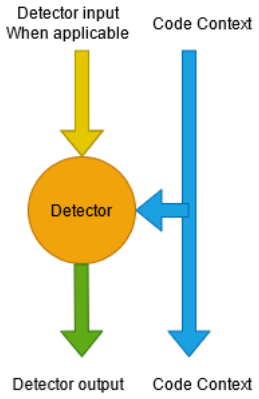

Figure 7.18: A atomic detector

In figure 7.18 an abstract design of a detector is presented. A detector is a list of query language statements that need to be executed in an execution space where the detector is provided with the proper input data. As an input, the detector needs the code context, which consists of the pseudo-AST representation of the source code. The input labeled "detector input when applicable" is used for cascading purposes of detectors.

DETECTOR OUTPUT/INPUT

It provides an output, which is a list of detected subjects or scopes and a copy of its input context [2]. Where the detector's findings occupy the last position in the list and where the input occupies the first position of the list.

[1] parallel detectors have not yet been necessary and are therefore not designed

[2] Providing this copy of the input to the output is an implementation choice to ease ongoing changes in detector requirements. The query language statements do not have to cope with this implementation choice.

Other input information that a detector needs, is a scopable object, this object is then used as a starting point. This can be any subject, as long as all information is provided. A detector that has to find all methods in a class, needs the class as a scopable subject. Hence, method?@classX. Some detectors need more information, like when a future signature of a to-be added method has to be detected in the source code context. Such a detector needs a scope and a future signature. The output of a detector is the context of the detected entity. A detector that detects for methods will output all pseudo-AST node references to these methods. An example output of a pseudo-AST reference to a detected method will be: **TestMeY@ClassY**@Package. A detector that detects if a method is still in use by callers, will output a list of pseudo-AST node references to that callers. Besides detector output, a verdict mechanism is added to the design to increase the flexibility of a detector as can be seen in picture 7.19.

VERDICT PROCESSING

The verdict mechanism consists out of decision logic, which combines detector findings into a final verdict. This is not very useful for atomic detectors, but it is very useful for compositions of detectors. Atomic detectors used in this thesis have their verdict logic build in the presence detector, which is more complex [1]. A composite detector exists out of multiple detectors and their results are combined into a final verdict. A composite detector can be reused for other detection purposes by only rearranging its decision logic.

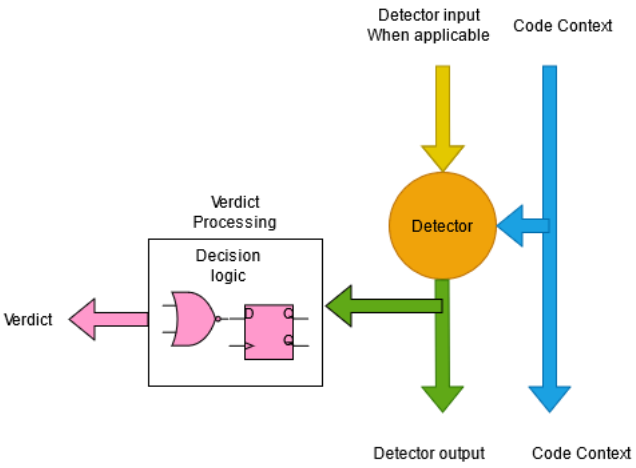

Figure 7.19: An atomic detector with verdict processing

Decision logic, present in the verdict processing mechanism, determines the quantities in the list of a detector's output. It determines whether there is no element, one element, or multiple elements present. An example of a verdict pseudo logic is presented below:

VERDICT1(**methodAllreadyDefinedInClass** ==!empty)
If the list from detector methodAllreadyDefinedInClass is NOT empty, then there will be a compiler error!

[1] This was a design choice at the start of this study

COMPOSITION OF DETECTORS

Detector compositions consist out of multiple detectors, but it has only one verdict processing mechanism. The verdict processing mechanism is rather specialistic, but it is highly reconfigurable and reusable. When designing detectors, one has to think about combining their output data as well, which is what a verdict processor does. The logic of the verdict processor has been replaced by a Look Up Table (LUT) with embedded logic. In this way, a verdict mechanism can be more easily reused and reprogrammed when necessary. A verdict mechanism should be scaleable to the number of detectors that it uses. A detector composition, therefore, exists out of:

- One or more detectors (these are the query language statements in an execution space.
- A verdict processor that scales with the number of detectors used, because the verdict processor also deserializes the detector findings to parallel entities [1].
- A connection from the output of the last detector to the verdict processor. This detector has all information of its previous detectors including the input context.
- A hazard- or verdict LUT which replaces the separate logic verdict statements.
- A connection between interconnected detectors. Except for the first and the last detector. These make cascading of composite detectors possible.
- Context input, like the current method signature and the future signature. This information must be available for all serialized detectors via a list of cascaded findings.

The composition of detectors can be seen in figure 7.20. This composition includes the verdict processor which produces the verdict and uses a LUT to generate the correct verdict. The detectors provide the information from its output to the others detector's input as can be seen in figure 7.21. The verdict processor is necessary to provide a verdict for a specific detector purpose and it must be redesigned when the purpose of the detector composition changes. This redesign comprehends:

- Scale the verdict processor towards the number of detectors[2].
- Redesign the verdict logic. Which must be performed by hand by a detector designer. This implicates changing and updating the LUT.

[1] An abstract design suggestion about how to communicate the detector's output to the verdict processor has been made. In this abstract design, this is done serially, but this does not have to be a must-have requirement for implementation

[2] because in this design assumption the detector data is serially transmitted

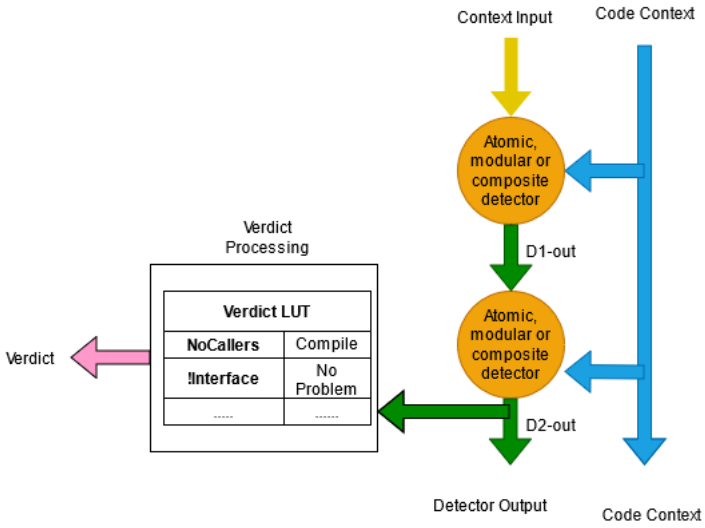

Figure 7.20: A composite detector with verdict processing

DETECTOR INPUT OUTPUT

The serialized output[1]. of the detectors is cascaded as can be seen in figure 7.21 and in figure 7.22.

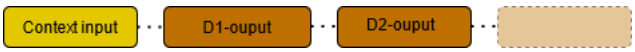

Figure 7.21: Serialized and cascaded output of all detectors

The separate data of all query language statement Aliases within the detector is also emitted in a serial manner, which implicates that all data is available for diagnostic- or for advisory purposes. This also implicates that each query language statement alias must always have a unique name.

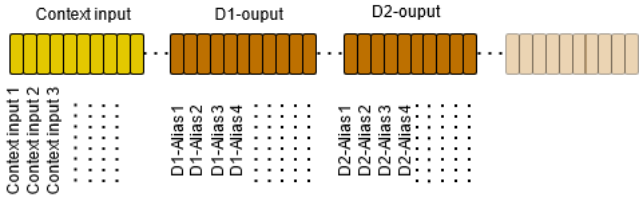

Figure 7.22: Serial serialized and cascaded output of all detectors

Detectors can be cascaded without change, however, not all detectors are sensitive to the same input context. If a detector cannot cope with the context input, like a detector that expects a class subject and receives a method signature, it should output a specific symbol that represents a fault condition. This prevents false positives[2]. Normally it should not receive abnormal data when properly designed.

[1] An abstract design suggestion about how to communicate the detector's output to the verdict processor has been made. In this abstract design, this is done serially, but this does not have to be a must-have requirement for implementation

[2] It is not yet known how to flag for fault conditions

COMPOSED DETECTORS

A composition of detectors is presented in figure 7.23. The verdict processors of the already created compositions are not used, however, they do remain present since they are a part of the original detector composition. For new composed compositions, a new verdict processor is necessary. In principle every detector can be cascaded, however, not all detectors are sensitive for a certain context.

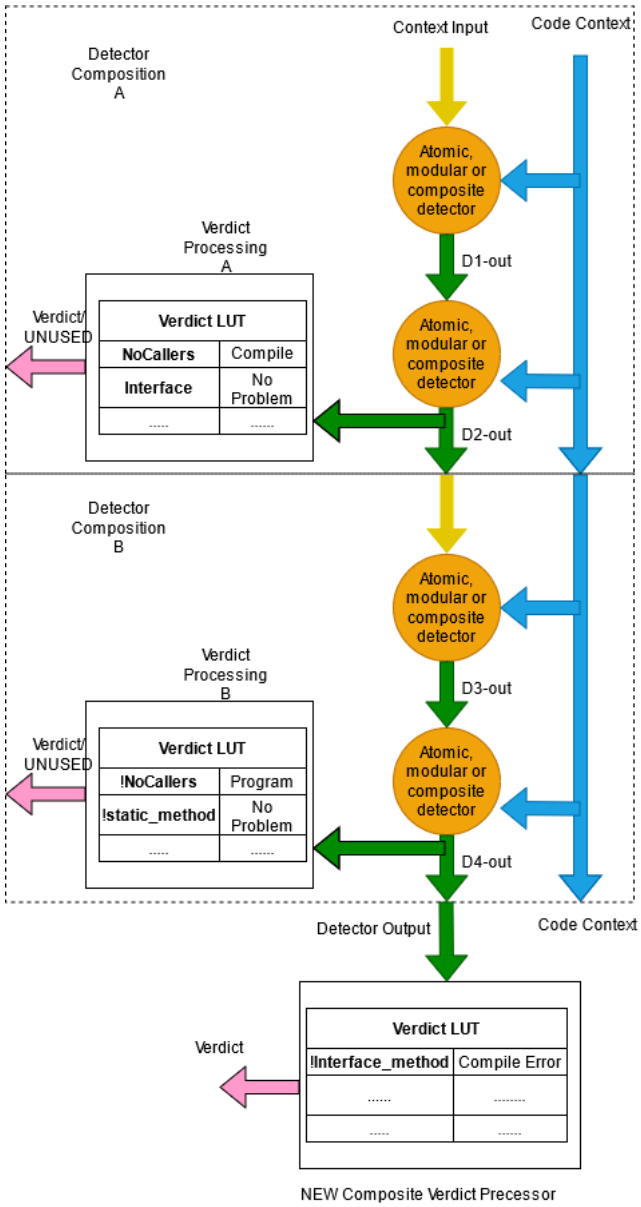

Figure 7.23: Serialized composition of composed detectors

A VERDICT LUT TO CREATE FLEXIBLE SWITCHING LOGIC

The verdict LUT combines a detector's output by using logic statements that determine the hazard. A typical LUT[1] for selecting hazards with logic is presented in figure 7.24.

Verdict LUT	
!InterfaceCaller	AM-H0
MethodAlreadyInClass	RC-H1-3
!CallerInSub && !StaticMethod	AA-H3

Figure 7.24: Lookup table with logic to select for hazard

The verdict LUT acts as a flexible logic switch between the detectors and the concerning hazard. Since the data from all detectors is available, the data can be used as a logic selection mechanism. Aliases used in query language statements can be used in the logic selection row. The output of a LUT can be seen on the right. This output selects the correct hazard.

[1] This verdict LUT is for demonstrational purposes only, references in this LUT are examples only.

7.3.2. How can multiple detected potential software harm be combined into final composite refactoring advice?

The final goal is to provide the refactorer with an advice. In this chapter, a flexible method to provide an advice is introduced. The mechanism from figure 7.23 is reused and extended with a LUT for an advice table, which is shown in figure 7.25.

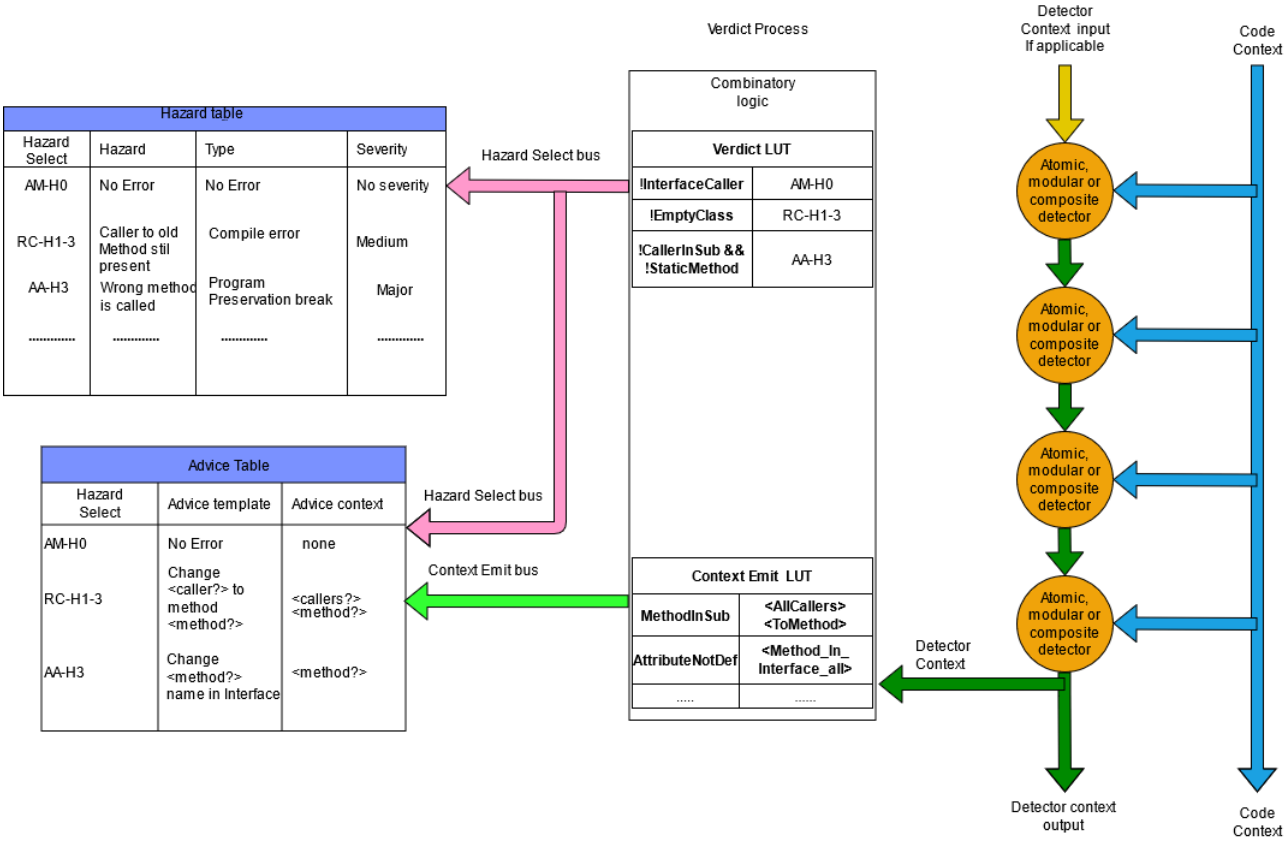

Figure 7.25: A composite detector with verdict processing

The hazard select bus provides a hazard selection mechanism, which is controlled by the verdict LUT. The verdict LUT [1] switches the correct hazard via hazard table [1]. The advice table is also switched via the hazard select bus. The advice table produces the correct advice with context parameters. These advice context parameters can be recognized by their bracket and a question mark.

Context emit LUT

The context emit LUT[1] provides for the correct detector context data towards the advice mechanism. The context emit LUT controlled separately from the hazard selection.

[1]The hazard table in figure 7.25, the advice table, the Verdict LUT, and the context emit LUT are for demonstrational purposes only, references in these tables are examples only.

Context Emit LUT	
MethodInSub	<AllCallers> <ToMethod>
AttributeNotDef	<Method_In_ Interface_all>
.....	.....

Figure 7.26: A context emit LUT

The right part of the context emit LUT can contain alias outputs of any query language statement that is present in the used detectors. This can be seen in figure 7.26. The context emit LUT is used to decouple for changes in the advice mechanism and for changes in the query statements [1].

Short	Context	Advice	Severity	Context variables
AC-H1	Class is already defined	Change name of class <class>	Low	<class?>

Table 7.30: Advice table for Add Class

THE ADVICE TABLES USE ALIASES FROM QUERY LANGUAGE STATEMENTS

The advice tables refer to the pseudo-AST subjects. An example advice table can be seen in table 7.30. If a context variable, like <class?> is described in an advice table, the advice mechanism should expect an alias to a query language statement that delivers class subjects as an output. The advice mechanism should therefore not be sensitive for wrong AST subjects. By using this flexible mechanism, advice tables context variables are decoupled from statement alias names, since both are applicable for change.

ADVICE TABLES FOR ALL KNOWN HAZARDS

The advice tables for all known hazards have been defined and can be seen in Appendix C of this document. These tables are organized by their hazards and provide for the hazard context. The tables provide for a hazard advice template with context variables which will be replaced by subjects from the context emit LUT. ie. The advice template "This class <class?> is in use." has context variable <class?> which will get replaced by the name of the class. The tables also provide for a severity tag, which makes it possible to prioritize hazards when there are multiple hazards defined. The coupling between the context variable and the aliases is performed by the context emit LUT.

[1] This motivates the abstract design suggestion about to serially communicate the detector's output to the verdict processor.

7.3.3. VALIDATION FOR THE MICROSTEP THEORY

The Rename method mechanics from fowler's refactoring mechanics will be used to validate the presence detection mechanism of hazardous code fragments.

The mechanics for Rename Method.

- Check to see whether this method signature is implemented by a superclass or subclass. If it is, carry out these steps for each implementation.
- Declare a new method with the new name. Copy the old body of code over to the new method and make any alterations to fit..
- Compile.
- Change the body of the old method so that it calls the new one.
    - *If you only have a few references, you can reasonably skip this step.*
- Compile and test.
- Find all references to the old method and change them to refer to the new one. Compile and test after each change.
- Remove the old method.
    - *If the old method is part of the interface and you cannot remove it, leave it in place and mark it as deprecated.*
- Compile and test

Figure 7.27: Rename method mechanics Fowler

When transformed to microsteps, the following microsteps are needed to replace the add and/or remove mechanics from 7.27:

- Microstep M1 - Add Method. with a new name and the old body.
- Refactoring step RS1: Change references from the old method to the new method.
- Microstep M2 - Remove Method, remove the old method.

Example 7.1: The mechanics transformed to microsteps

7.3.4. VALIDATION FOR THE DETECTOR THEORY

The code example in figure 7.28 will be used to test the mechanisms.

```
class ClassX
{
	public void testMeX(int justaparamX)
		{
			if(justaparamX<=0) justaparamX=100;
			for(int a=0;a<justaparamX;a++) {
					System.out.println("Hundred times " +a);
			}
		}
	public void anotherMethod() {
		testMeX(100);
		}
}
```

Figure 7.28: Rename method test code

The microstep transformation of example 7.1 will be applied to this test code in figure 7.28. The refactoring activity to be performed is a rename method on the method testMeX in ClassX. The new method name will be "testMeY"

Before this action takes place, the Add Method AM-H hazards will be tested. These hazard tables can be found in Appendix C. The hazards with a severity beyond Low will be tested first, which is AM-H2. Hazards with the same severity can be tested in random order.

To test for hazard AM-H2, the detector which exists out of query language statements AM-H2-HCPD has to be executed. This detector delivers an empty list which equals !CallSubToSiginSupNoKeyAM, since there are no superclasses.

The next hazard is AM-H1, which executes AM-H1-HCPD. This detector delivers an empty list, which equals !MethodAlreadyDefinedInClass, since there is no other method in this class with the name "TestMeY".

No advice mechanism is triggered since there are no matches in the LUT entries, so no hazard and no advice is selected. The Add method LUTs tables can be found in Appendix D in Tables 31 and 32.

Since all detectors for the M1 add Method microstep passed, the Add Method microstep with a new name and old body can be applied safely without software harm. This temporarily creates similar methods with different names.

```
class ClassX
{
	public void testMeY(int justaparamX)
	{
		if(justaparamX<=0) justaparamX=100;
		for(int a=0;a<justaparamX;a++) {
					System.out.println("Hundred times " +a);
		}
	}

	public void testMeX(int justaparamX)
		{
			if(justaparamX<=0) justaparamX=100;
			for(int a=0;a<justaparamX;a++) {
					System.out.println("Hundred times " +a);
			}
		}
	public void anotherMethod() {
		testMeX(100);
		}
}
```

Figure 7.29: Rename method test code

The refactoring step RS1 is executed and the caller testMeX(100) to the old method in method anotherMethod() is changed to call the new method testMeY(100). The microstep M2 will be performed, but before this microstep is performed the Remove Method RM-H Hazards will be tested for. The major severity hazards will be tested first, which is RM-H2 for the Remove Method microstep (Appendix C).

RM-H2 executes the RM-H2-HCPD which detects hazardous code fragment presence. This detector yields an empty list **!CallSubToSiginSupNoKeyAM**, since there are no superclasses. The remaining hazards of the same severity level can be tested in random order.

RM-H3 executes the RM-H3-HCPD, which yields an empty list **!MethodStatDefDefInInterfAM** since there is no interface.

RM-H1 executes RM-H1-HCPD, which yields an empty list. The verdict LUT and the context emit LUT are not triggered. The verdict LUT does not select a hazard on the hazard select bus and the context emit LUT does not emit context towards the advice mechanism.

Since all detectors for the M2 remove Method microstep passed, the Remove Method microstep can be applied safely without software harm.

This leads to the following result in figure 7.30

```
class ClassX
{
	public void testMeY(int justaparamX)
	{
		if(justaparamX<=0) justaparamX=100;
		for(int a=0;a<justaparamX;a++) {
					System.out.println("Hundred times " +a);
		}
	}

	public void anotherMethod() {
		testMeY(100);
		}
}
```

Figure 7.30: Final result after refactoring

To further demonstrate that the mechanism works, assume the hypothetical situation where The M1 microstep succeeded successfully, but the refactorer forgot to perform refactoring step RS1, which comprehends to change the caller in method anotherMethod() from testMeX(100) to the new method testMeY(100). So that this caller is incorrect.

What happens is that RM-H1 executes RM-H1-HCPD, which yields an AST reference to the caller testMeX(100); in the method "anotherMethod(). The verdict LUT and the context emit LUT (table 33 Appendix D) are triggered by **FindCallersToSignature2BRemoved**. The verdict LUT selects the RM-H1 hazard on the hazard select bus. The context emit LUT emits the alias **FindCallersToSignature2BRemoved** towards the advice mechanism.

The advice mechanism renders the hazard and the advice:
"Callers to method became incorrect after method change/removal"
The advice is to:
"Change caller(s) TestMeX(100)" [1].

RM-H2 has to be detected, so RM-H2-HPCD is executed which yields an empty list since there is no inheritance tree. No extra advice is rendered.

RM-H3 has to be detected, so RM-H3-HPCD is executed which yields an empty list since there is no interface. No extra advice is rendered.

The refactorer now corrects the error and repeats the M2 microstep until the advice mechanism no longers produces an advice or flags for issues.

[1] During the implementation of the advice mechanism, this advice should be expanded with extra information, like "change caller(s) TestMeX(100) in method "anotherMethod" on line xxx.

8

CONCLUSION

The main research question was divided into six partial questions.

"How can we provide for a refactoring advice based on: the performed refactoring activity, and the source code under refactoring?"

The subquestions were:

RQ1: ***How can a hazardous code fragment be described in an easy to comprehend manner?***

The specially designed query language based on the pseudo-AST representation created a possibility to describe code relationally. In chapter 7.1 a flexible and comprehensible method was described how to create hazardous code pattern descriptions with the help of the query language.

RQ2: ***How can we transform Fowler's refactoring mechanic steps into microsteps?***

The microsteps were derived from Fowler's mechanics without issues. Chapter 7.2.1 showed that most changes directed from the mechanics implicate a cascade of remove- and add microstep action. By reducing the steps into more atomics microsteps, a more detailed hazard detection can be performed and a better advice can be provided.

RQ2.1: ***What will the hazard(s) of a particular microstep be?***

In chapter 7.2.2, the hazards were derived on a microstep level and archived in hazard tables accompanied with their microstep. The coverage of the hazards is not complete and is an ongoing task since not all hazards were known during this research.

RQ2.2: ***Which hazardous code pattern descriptions are needed for each microstep's induced hazard to detect potential software harm?***

In chapter 7.2.3 the query language has proven to be very powerful in creating hazardous code pattern descriptions to detect the presence of hazardous code fragments in the source code entity. Hazardous code pattern descriptions can be expressed in a very precise and structured manner that resembles the use of formulas in mathematics. The hazardous code pattern descriptions are linked to a trigger event in separate tables for each microstep. This collection of harmful trigger events will be used as detectors in the advice mechanism. The harmful-trigger-event is triggered by the to-be-performed microstep.

RQ3: ***How can hazardous code fragments be detected in a source code entity flexibly?***

The hazardous code pattern descriptions that are part of a detector, can be cascaded in a loosely coupled manner. Chapter 7.3.1 showed that the data of each detector is transmitted to the next detector, where it can be used as input data and is transparently transmitted through each detector towards the verdict LUT. This mechanism decouples the decision mechanism from the collection of facts. The serial transmission of hazardous code presence detector data is an abstraction, which means that any form of data transmission is suitable during implementation, as long as all hazardous code presence detector data is available for the verdict mechanism. Due to the separation of fact collection and decision making, the reuse of detectors is strongly simplified. The verdict LUT also eases the creation of future detector design via no-code or low-code tools.

RQ3.1: ***How can multiple detected potential software harm be combined into a final composite refactoring advice?***

The detector is expanded with an advise rendering mechanism. This mechanism consists out of a context emit LUT an advice table and a hazard table. The context emit LUT decouples the advice context selection from the verdict mechanism. Chapter 7.3.2 showed that this design feature has the purpose to decouple the changes in the hazardous code presence detectors and the advice mechanism. The advice tables are organized based on microstep induced hazards. The hazard tables and the Advice tables are separated to decouple for changes in the advice layout and for changes in the hazard definitions. Multiple detectors can be created from just one composed detector, just by changing their verdict- and context emit LUT, the advice tables, and the hazard tables. A composite advice is created by cascading the correct presence detectors, selecting the correct facts from the presence detectors, and an entry in the advice- and hazard tables. By using this method, multiple detected potential software harm can be combined into a final refactoring advice in a highly flexible manner.

9

RECOMMENDATIONS AND FUTURE WORK

This chapter will elaborate on the subjects which require further study.
Future research directions and recommendations:

- **Refactoring does not necessarily prevent architectural decay**: Fowler's refactoring mechanics have the predicate to be behavior preserving, which can be seen in chapter 5.1.1. Should refactoring be conservating the software architecture as well? This could be another predicate that is above the predicate behavior preserving. This means another orchestration mechanism on top of the presented methods from this study.
- **Enhance the advice mechanism**: The advice mechanism in this thesis is quite advanced, however, it still misses some functionality. The detectors collect facts from the AST representation and communicate these facts towards the advice mechanism. However, for absolute flexibility, it should be possible to collect new facts based on the facts offered by the detectors. For example: when the detectors detect unchanged callers, the detector communicates a list of AST nodes that represent these callers. For the advice mechanism, it should be possible to determine if these callers are in a method with a certain name. As it is currently designed, these facts have to be communicated by the detectors, which means that presenting new facts in a refactoring advice means new functionality in the detectors. This coupling should be reduced by a fact investigator mechanism, which is in between the detectors, the advice mechanism and has access to the code context. A diagram of a design can be found in appendix F .7.5.
- **The query language needs to be maintained**: The query language needs to be kept up to date and it needs to be maintained.
- **Make the advice mechanism fault-tolerant**: There is no fault detection in the advice mechanism as it is currently designed. An empty list can mean that there was no detection or it means a faulty designed detector. Or wrong data is used for a refactoring advice. A correctly designed detector should not produce errors.
- **Enhancement of the list of Hazardous Trigger Events**: New hazardous code pattern descriptions and possibly new microsteps will be needed. This list should be expanded as necessary.

- **Find a fast method to generate an AST representation**: Since the performance of the designed method is depending on the AST representation, this mechanism should be as fast as necessary.
- **Find a method to link the AST representation to the source code**: It is very comfortable to work in an IDE where an advice refers to line numbers. A method should be found to link the AST representation to the source code.
- **Provide a low code solution for composing detectors**: The use of low-code (or no-code) solutions with graphical drag and drop to construct detectors should be the aim. In this way, there is no threshold to make new compositions and/or new detectors without vast (programming) knowledge about the inner workings of the detectors. Making them easy to modify, and provide for easy reuse is the aim.
- **Provide a working and distributable prototype**: A working and distributable prototype should be made to make sure that others can learn from it. The code should be architecturally correct to ease understanding. Maintaining good documentation on the sourcecode is mandatory. The prototype should respect the architecture provided in this thesis since it offers modular implementation.
- **Real-time tracking of hazards** The provided mechanism in this thesis should be capable of tracking hazards in real-time. The aliases in the presence detectors represent pre-calculated lists, which has been a design choice. It is possible to enhance the presence detectors with a trigger mechanism so that the detectors can provide data on a run-basis. This trigger mechanism preferably detects for semantical or syntactical correctness of the code first.

BIBLIOGRAPHY

Patrick de Beer. Code Context based Generation of Refactoring Guidance, 2019. Graduation Assignment. 5, 7

Martin Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA, 1999. ISBN 0-201-48567-2. 8

Alfonso Fuggetta and Elisabetta Di Nitto. Software process. In Proceedings of the on Future of Software Engineering, FOSE 2014, page 1–12, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328654. doi: 10.1145/2593882.2593883. URL https://doi.org/10.1145/2593882.2593883. 4

ISO-14971. ISO/IEC 14971: Basic Concepts – Hazard, Hazardous Situation and Harm. September 1998. URL http://webstore.ansi.org/ansidocstore/product.asp?sku=ISOhttp://webstore.ansi.org/ansidocstore/product.asp?sku=ISOhttp://www.iso.ch/cate/d25845.html;https://webstore.ansi.org/. Available in electronic form for online purchase at http://webstore.ansi.org/ and http://www.cssinfo.com/. 15

William F. Opdyke. Refactoring Object-Oriented Frameworks. PhD thesis, USA, 1992. 10

S. Stuurman. Design for Change. PhD thesis, June 2015. 31

T. Tourwé and T. Mens. A survey of software refactoring. IEEE Transactions on Software Engineering, 30(02):126–139, feb 2004. ISSN 1939-3520. doi: 10.1109/TSE.2004.1265817. 4

APPENDIX A - REFACTORING TOOL

In research done by Patrick de Beer, an automated refactoring tool was presented which was using a solution based on a RAG. A RAG is capable of producing readable advice during refactoring. To do so, the RAG combines the source code context, advice templates, and decision logic into a single directed graph. A typical RAG can be seen in figure 1.

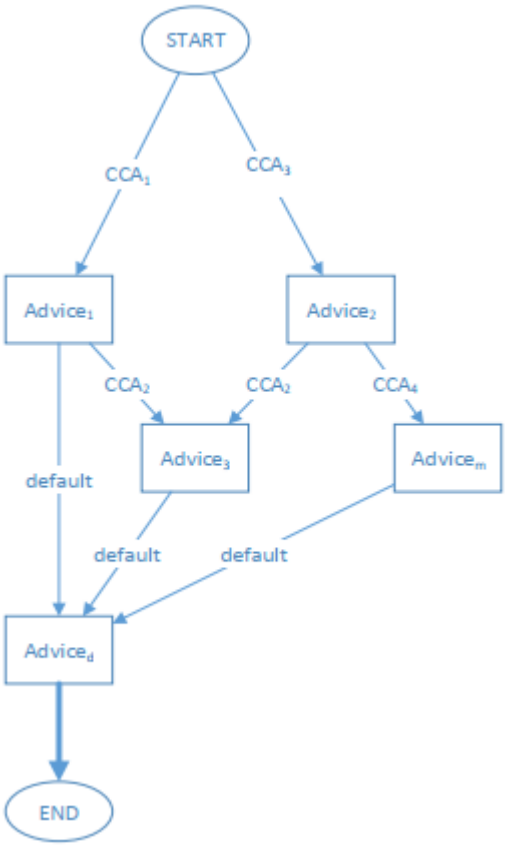

Figure 1: An extract-method RAG

In figure 1, a directed graph is seen with vertices, so-called Code Context Advice (CCA)'s, and readable advice (square boxes). The Advice templates are concatenated in order of a specific path through the graph. That specific path, or route, depends on the outcome of the CCA's. If a CCA is not valid, an alternative path must be present in the RAG. To prevent deadlock issues in the implementation, an alternative path is often constructed by using the negated CCA of the primary path, combined with empty advice. The vertice with the negated CCA and the empty advice holds no action and produces no artifacts. The RAG

consumes three input parameters, which are only used by the CCA's in the RAG. The output of the RAG is a concrete and readable advice. The input parameters for a RAG and thus for a CCA are:

- ***The refactoring subject***.
- ***The code context***.
- ***A Code Context Property Detector (CCPD)***.
  The CCPD evaluates the code context of the refactoring subject for the presence of a specific Code Context Property (CCP). A typical example CCP could be that a method with a signature is declared only once in all class definitions.

The *CCA* produces concrete advice from these inputs. The CCPD will detect the presence of the CCP within the code context. From these facts, the advice is rendered by the CCA with the help of an instantiator which assists in adding concrete values based on keywords from the advice template provided by the CCPD. A conceptual representation of a CCA can be seen in figure 2.

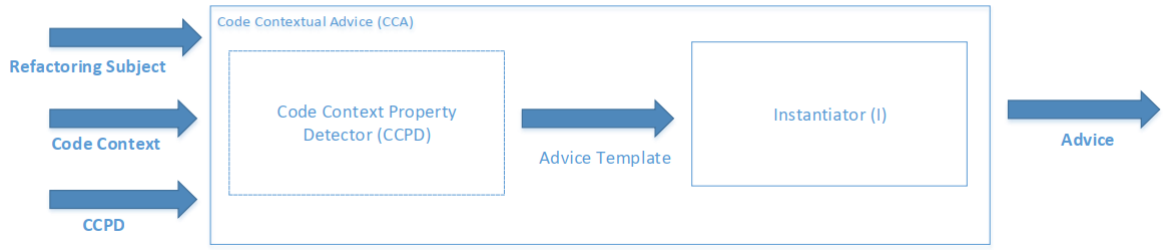

Figure 2: A CCA

The CCPD consumes the refactoring subject and the code context and generates an advice template from it. To be more precise, a CCPD detects a CCP in the code context and creates advice (template) from these parameters as can be seen in figure 3. The CCP's are mapped to an advice template that describes the risk associated with the refactoring action and the code context.

Figure 3: A CCPD

A CCP mapped advice derives risks caused by refactoring action(s) for a specific code context and provides solutions and possible alternatives. This differs from Opdyke's danger avoidance approach. A typical CCP mapped advice can be seen in table 1 where the danger caused by a refactoring risk is explained and a possible solution is raised.

CCP	Advice
NameHiding - Local variable hides class field	The code you extract contains variables X/Y/Z that are hiding fields in your class. Start by renaming your variables. This will prevent those variables in your extracted code will use accidentally class fields i.s.o. the local variable. The compiler will happily compile if you forget them.
SingleArgument - Single argument must be passed to extracted code	Variable X is used in your new method and assigned a value before your method is called. Copy argument declaration for ArgumentX as an input parameter into your new method

Table 1: A subset of the list of CCP's mapped to advice (Risks).

Typical CCP's were described by Patrick de Beer in his research. Each specific refactoring needs a specific set of CCP's. For example, if a rename method refactoring would be performed, the refactorer must know if either the method to be renamed was defined only once, or multiple times. Discovering these method definitions, are typical CCP's. Patrick de Beer demonstrated that two different refactorings use different sets of CCP's. In these presented sets, the CCP's per refactoring are unique, which can be seen in figures 4 and 5.

CCPD function	Description	Context Detector
NH	Local variable hides class field	MethodExtractNameHiding
ZA	Zero arguments must be passed to extracted code	MethodExtractNoneArguments
SA	Single argument must be passed to extracted code	MethodExtractSingleArgument
MA	Multiple arguments must be passed to extracted code	MethodExtractMultipleArgument
ZR	Extracted code should return no result	MethodExtractNoneResults
SR	Extracted code should return single result	MethodExtractSingleResult
MR	Extracted code should return multiple results	MethodExtractMultipleResult
CRet	Conditional return present in extracted code	MethodExtractControlReturn

Figure 4: A CCP extract method set.

CCPD function	Description	Context Detector
SiD	Method declared only once	MethodSingleDeclaration
MD	Methods with same signature in class hierarchy	MethodMultipleDeclaration
ID	Method declared in public interface	MethodInterfaceDeclaration
SD	Method overrides a method of one of the super classes	MethodOverride
MO	Method overloaded in same class	MethodOverload
MA	Method can have @Override annotation	MethodOverrideNoAnnotation

Figure 5: A CCP rename method set

A typical, although quite complex RAG can be seen in figure 6. This RAG's purpose is to provide advice while performing an extract method refactoring. The negated vertex $CCA\langle\neg NH\rangle$ in the top of the RAG is used to make the software implementation of the RAG easier, by preventing deadlock or timelock issues. The CCP's for this RAG can be seen in figure 4. The RAG presented by Patrick has an interesting vertex worth looking at. The vertex

CCA $\langle ZA \rangle$ (empty) CCA $\langle ZR \rangle$ detects for methods with no arguments and not returning any values. A situation, where a method without arguments can return a value, like refactoring the method with the definition **double givePI_asNumber(void)**, would in this RAG lead to a timelock or deadlock when implemented as software. This illustrates how complex an automated advice mechanism for a rather simple refactoring action can become.

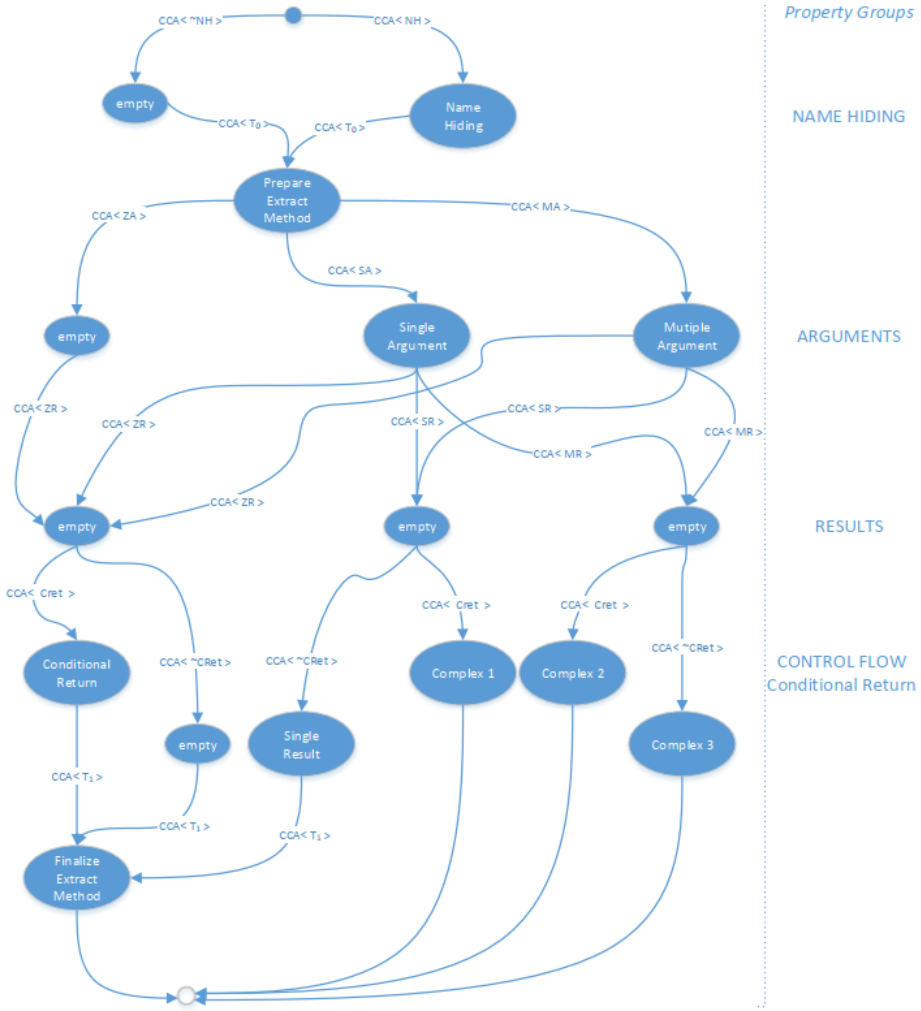

Figure 6: An extract-method RAG

APPENDIX B - ADD A PARAMETER TO A METHOD

```
public class x
{
   public int test(int justaparam)
         {
         system.out.println("This is behavior preserving \n" + justaparam);
         }
}

public class y extends x
{
   public int test(void)
          {
          system.out.println("We add a parameter to this method \n");
          return(0);
          }


   public void runner()
           {
           test(5);     // calls the superclass method test with
           }            // parameter int, because of inherritance.
}
```

Figure 7: add parameter to method example before refactoring

We apply the mechanics from fig 8, from Fowler to this code example in figure 7.

The mechanics of *Add Parameter* are very similar to those of Rename Method.

- Check to see whether this method signature is implemented by a superclass or subclass. If it is, carry out these steps for each implementation.
- Declare a new method with the added parameter. Copy the old body of code over to the new method.
    - *If you need to add more than one parameter, it is easier to add them at the same time.*
- Compile.
- Change the body of the old method so that it calls the new one.
    - *If you only have a few references, you can reasonably skip this step.*
    - *You can supply any value for the parameter, but usually you use null for object parameter and a clearly odd value for built-in types. It's often a good idea to use something other than zero for numbers so you can spot this case more easily.*
- Compile and test.
- Find all references to the old method and change them to refer to the new one. Compile and test after each change.
- Remove the old method.
    - *If the old method is part of the interface and you cannot remove it, leave it in place and mark it as deprecated.*
- Compile and test

Figure 8: The mechanics for adding a parameter to a method.

The steps of the mechanics marked in red "declare a new method with the added parameter" and " Find all references to the old method and change them to refer to the new one" is not program behavior preserving. Adding the method in the subclass will cause redefinition and the call to the method test(int) in the superclass will now call to the newly redefined method in the subclass. The refactorer does not know that this method call refers to the method in the superclass. Changing the old reference and change it to the new method, will create a double fault in this example. Both faults create program behavior preservation problems 8.

```
public class x
{
   public int test(int justaparam)
         {
         system.out.println("This is behavior preserving \n" + justaparam);
         }
}

public class y extends x
{
//  public int test(void)       // Removed, see mechanics
//         {
//         test(-99);           // odd value, see mechanics.
//         }

   public int test(int newparam)
         {
         system.out.println("Preservation break happens if this is printed. \n" + n
         return(0);
         }

   public void runner()
          {
          test(5);             // Calls subclass method test with
          }                    // parameter int, because local redefinition
                               // It calls the new changed method in subclass and
                               // fowler mechanics direct to call the newly changed
                               // and not the superclass method. So it is a double
}
```

Figure 9: Remove parameter from method example after refactoring.

APPENDIX C - FULL-BLOWN AST REPRESENTATION OF THE CODE FRAGMENT EXAMPLE IN FIGURE 5.4

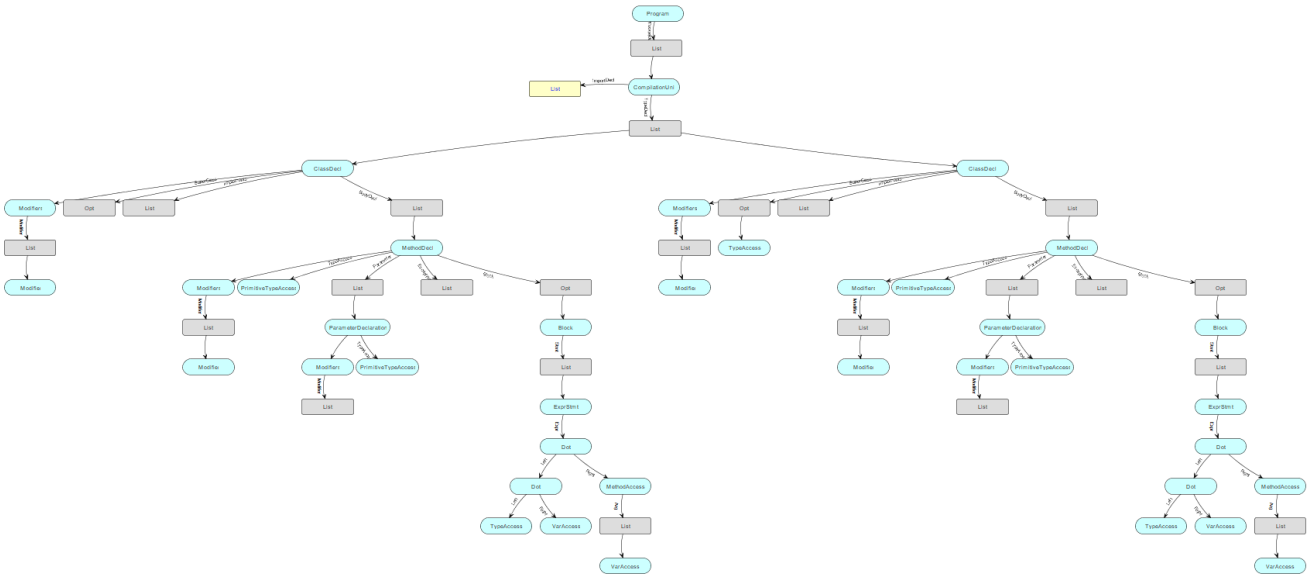

Figure 10: Full-blown AST representation of the code fragment example from figure 5.4 derived with ExtendJ Explorer.

APPENDIX C - ADVICE TABLES FOR MICROSTEP INDUCED HAZARD

For the Add Class hazards, the advice table is presented in table 2. The <class?> context is provided by the detector via the Context Emit LUT.

.1. ADVICE TABLES

.1.1. ADD CLASS

Short	Context	Advice	Severity	Context variables
AC-H1	Class is already defined	Change name of class <Class?>	Low	<class?>

Table 2: Advice table for Add Class

.1.2. REMOVE CLASS

For the Remove Class hazards the advice table is presented in table 3. The <Instantiation?>, <Class?>, <Field?>, <Caller?> context is provided by the detector via the Context Emit LUT.

Short	Context	Advice	Severity	Context variables
RC-H1-1	Class still instantiated	Remove instantiation <Instantiation?> or don't remove class <Class?>	Low	<Instantiation?> <Class?>
RC-H1-2	Class still referenced	Remove field reference <field?> or don't remove class	Low	<Field?>
RC-H1-3	Caller to method in class	change caller <Caller?>	Low	<Caller?>
RC-H2-1	ChildClass is part of inherritance tree	Do not rename <class?>	Low	<Class?>
RC-H2-2	Superclass is part of inherritance tree.	Do not rename <class?>	Low	<Class?>
RC-H3	Static method, Inner static class still called by caller	Static Method referenced by <caller?> is still being used. Move static method to separate class or do not rename class or move functionality towards use	Low	<Caller?>
RC-H4	Class is renamed possible design pattern broken	do not rename abstract class <class?>	Low	<Class?>

Table 3: Advice table for Remove Class

.1.3. RENAME CLASS

For the Rename Class hazards, the advice table is presented in table 4. This table is a combination of the remove and the add microstep hazards. The <Instantiation?>, <Class?>, <Field?>, <Caller?> context is provided by the detector via the Context Emit LUT.

Short	Context	Advice	Severity	Context variables
RNC-H1-1	Old class still instantiated	Rename instantiation <Instantiation?> or don't rename class <Class?>	Low	<Instantiation?> <Class?>
RNC-H1-2	Class still referenced	Rename field reference <field?> or don't rename class	Low	<Field?>
RNC-H1-3	Caller to method in class	change caller <Caller?>	Low	<Caller?>
RNC-H2-1	ChildClass is part of inherritance tree	Do not rename <class?>	Low	<Class?>
RNC-H2-2	Superclass is part of inherritance tree.	Do not rename <class?>	Low	<Class?>
RNC-H3	Static method, Inner static class still called by caller	Static Method referenced by <caller?> is still being used. Move static method to separate class or do not rename class or move functionality towards use	Low	<Caller?>
RNC-H4	Class is renamed, possible design pattern broken	do not rename abstract class <class?>	Low	<Class?>
RNC-H5	New name of class is already present	Please use other name for <class?>, since that name is in use	Low	<Class?>

Table 4: Advice table for Rename Class

.1.4. ADD ATTRIBUTE

For the Add Attribute hazards, the advice table is presented in table 5. The <field?> context is provided by the detector via the Context Emit LUT.

Short	Context	Advice	Severity	Context variables
AA-H1	Attribute already defined	Use other attribute name for <Field?>	Low	<Field?>
AA-H2	Inner-/outer class attribute redefinition (Attribute from outer class is used in inner class and a new attribute with the same name and type is added to inner class)	Use another attribute name for <Field?>	Major	<Field?>
AA-H3	An inheritance redefined attribute in Subclass (Attribute from superclass is used in sub class and new attribute with same name and type is add to subclass)	Use another attribute name for <Field?>	Major	<Field?>

Table 5: Advice table for Add Attribute

.1.5. REMOVE ATTRIBUTE

For the Remove Attribute hazards, the advice table is presented in table 6. The <Fieldassignment?> context is provided by the detector via the Context Emit LUT.

Short	Context	Advice	Severity	Context variables
AA-H1	Not all users of attribute are aware of removal	Change assignment <Fieldassignment?>	Low	<Fieldassignment?>
AA-H2	Inner-/outer class attribute redefinition (when removed from inner)	Change inner attribute name for assignment <Fieldassignment?> in inner class.	Major	<Fieldassignment?>
AA-H3	Inherritance redefined attribute in Subclass (Superclass Attribute used after removal)	Use other attribute name for assignment <Fieldassignment?> in subclass	Major	<Fieldassignment?>

Table 6: Advice table for Remove Attribute

.1.6. RENAME ATTRIBUTE

For the Rename Attribute hazards, the advice table is presented in table 7. This table is a combination of the remove and the add microstep hazards. The <Fieldassignment?>, <Field?> context is provided by the detector via the Context Emit LUT.

Short	Context	Advice	Severity	Context variables
RNA-H1	Not all users of attribute are aware of removal	Update assignment(s) <Fieldassignment?> to <field?>	Low	<Fieldassignment?> <field?>
RNA-H2	Inner-/outer class attribute redefinition (when removed from inner)	Change attribute name for attribute <Field?> in inner class and update fieldassignment <fieldassignment?> to that new name.	Major	<Field?> <Fieldassignment?>
RNA-H3	Inherritance redefined attribute in Subclass (Superclass Attribute used after removal)	Change attribute name for attribute <Field?> in inner class and update fieldassignment <fieldassignment?> to that new name.	Major	<Field?> <Fieldassignment?>
RNA-H4	Attribute already defined	Attribute <field?> is already defined at <field?>	Low	<Field?> <Field?>

Table 7: Advice table for Rename Attribute

.1.7. ADD METHOD

For the Add Method hazards, the advice table is presented in table 8. The <Caller?>, <Method?> context is provided by the detector via the Context Emit LUT.

Short	Context	Advice	Severity	Context variables
AM-H1	Method already defined	Change method name <Method?> to another name	Low	<Method?>
AM-H2	Inherritance Redefined Method in Subclass. caller from subclass calls Subclass method used after adding a parameter to subclass method, instead of superclass method.	Change method <Method?> name	Major	<Method?>

Table 8: Advice table for Add method

.1.8. REMOVE METHOD

For the Remove Method hazards, the advice table is presented in table 9. The <Caller?>, <Method?> context is provided by the detector via the Context Emit LUT.

Short	Context	Advice	Severity	Context variables
RM-H1	Callers to method became incorrect after method change/removal	change callers <Caller?>	Low	<Caller?>
RM-H2	Inherritance Redefined Method in Subclass (Superclass method used after removal) .	Change method <Method?> name	Major	<Method?>
RM-H3	Method defined in interface	redefiniton of method <Method?> do not remove	Low	<Method?>

Table 9: Advice table for Remove method

.1.9. Rename Method

For the Rename Method hazards, the advice table is presented in table 10. This table is a combination of the remove and the add microstep hazards. The <Caller?>, <Signature?>, <Method?> context is provided by the detector via the Context Emit LUT.

Short	Context	Advice	Severity	Context variables
RNM-H1	Callers to method become incorrect after method rename	change callers <Caller?> to call new method with signature <Signature?>	Low	<Caller?> <Signature?>
RNM-H2	Method already defined	Change method name <method?> because it collides with <method?> to another name	Low	<method?> <method?>
RNM-H3	Inherritance Redefined Method in Subclass (Superclass method used after rename) .	Change method <Method?> name and change caller <Caller?> from subclass to call to new method name	Major	<Method?> <Caller?>
RNM-H4	Method defined in interface	redefiniton of method <Method?> or interface contract prohibits change	Low	<Method?>

Table 10: Advice table for Rename method

.1.10. Add Interface

For the Add Interface hazards, the advice table is presented in table 11. The <Interface?> context is provided by the detector via the Context Emit LUT.

Short	Context	Advice	Severity	Context variables
AI-H1	Interface already defined	change interface <Interface?> name	Low	<Interface?>

Table 11: Advice table for Add Interface

.1.11. REMOVE INTERFACE

For the Remove Interface hazards, the advice table is presented in table 12. The <Interface?> context is provided by the detector via the Context Emit LUT.

Short	Context	Advice	Severity	Context variables
RI-H1	Interface still in use by class	Do not remove interface <Interface?> or do not use interface contract.	Low	<Interface?>
RI-H2	Interface still in use by other interface	Move interface <Interface?> to other interface, or do not remove.	Low	<Interface?>
RI-H3	Interface holds static/default implementations	Move implementations in <Interface?> towards use.	Low	<Interface?>

Table 12: Advice table for remove interface

.1.12. RENAME INTERFACE

For the Rename Interface hazards, the advice table is presented in table 13. This table is a combination of the remove and the add microstep hazards. The <Interface?>, <Class?>, <Method?>,<Param?> context is provided by the detector via the Context Emit LUT.

Short	Context	Advice	Severity	Context variables
RNI-H1	Interface still in use by class	Do not rename interface <Interface?> or change implements in class <Class?> to <Interface?>.	Low	<Interface?> <Class?> <Interface?>
RNI-H2	Interface still in use by other interface	Move interface <Interface?> to other interface, do not remove ore rename implements in interface <Interface?>.	Low	<Interface?> <Interface?>
RNI-H3	Interface holds static/default implementations	Move <method?> implementations towards use.	Low	<method?>
RNI-H4	Interface already defined	Change interface <interface?> name	Low	<Interface?>

Table 13: Advice table for rename interface

.1.13. ADD PARAMETER

For the Add Parameter hazards, the advice table is presented in table 14. The <Caller?>, <Method?> context is provided by the detector via the Context Emit LUT.

Short	Context	Advice	Severity	Context variables
AP-H1	Callers to method became incorrect after adding parameter to method.	change callers <Caller?>	Low	<Caller?>
AP-H2	Method already defined after adding a parameter	Change method name <Method?> to another name	Low	<Method?>
AP-H3	caller from subclass calls Subclass method instead of superclass method, after adding a parameter to the method in the subclass.	Change method <Method?> name in subclass	Major	<Method?>
AP-H4	Method definition in interface. Interface contract broken after adding a parameter.	deprecate old method and create new method with body of old method for <Method?>	Low	<Method?>
AP-H5	Parameter with same name already exists.	Change parameter name of <Param?>	Low	<Param?>
AP-H6	Field with same name is already in use in method.	Change parameter name of <Param?>	Low	<Param?>

Table 14: Advice table for Add Parameter

.1.14. REMOVE PARAMETER

For the Remove Parameter hazards, the advice table is presented in table 15. The <Caller?>, <Method?>, <Fieldaccess?> context is provided by the detector via the Context Emit LUT.

Short	Context	Advice	Severity	Context variables
RP-H1	Callers to method became incorrect after removing a parameter from method.	change callers <Caller?>	Low	<Caller?>
RP-H2	Method already defined after removing a parameter from method	Change method name <Method?> to another name	Low	<Method?>
RP-H3	caller from subclass calls Subclass method instead of superclass method, after removing a parameter from the method in the subclass.	Change method <Method?> name in subclass	Major	<Method?>
RP-H4	Method definition in interface. Interface contract broken after removing a parameter.	deprecate old method and create new method with body of old method for <Method?>	Low	<Method?>
RP-H5	Parameter is still in use in Method.	Rename fieldaccess <fieldaccess?> in method <method?>	Low	<Fieldaccess?> <Method?>

Table 15: Advice table for Remove Parameter

.1.15. RENAME PARAMETER

For the Rename Parameter hazards, the advice table is presented in table 16. This table is a combination of the remove and the add microstep hazards. The <Caller?>, <Signature?>, <Method?> context is provided by the detector via the Context Emit LUT.

Short	Context	Advice	Severity	Context variables
RNP-H1	one or two parameter names are identical	change new parameter name <param?>	Low	<Param?>
RNP-H2	Parameter name is already in use inside method	Change parameter name <param?>	Low	<Param?>
RNP-H3	Cannot rename parameter since old parameter is still referenced inside method.	Change parameter name <param?>	Low	<Param?>

Table 16: Advice table for Rename Parameter

APPENDIX D - DETECTOR RECIPES FOR DETECTORS FOR SINGLE MICROSTEPS

The detector recipes comprehends filling tables with the correct information and providing a hazardous code presence detection algorithm designed with the query language.
For all "atomic" detectors, detector type A is used, see figure 11.

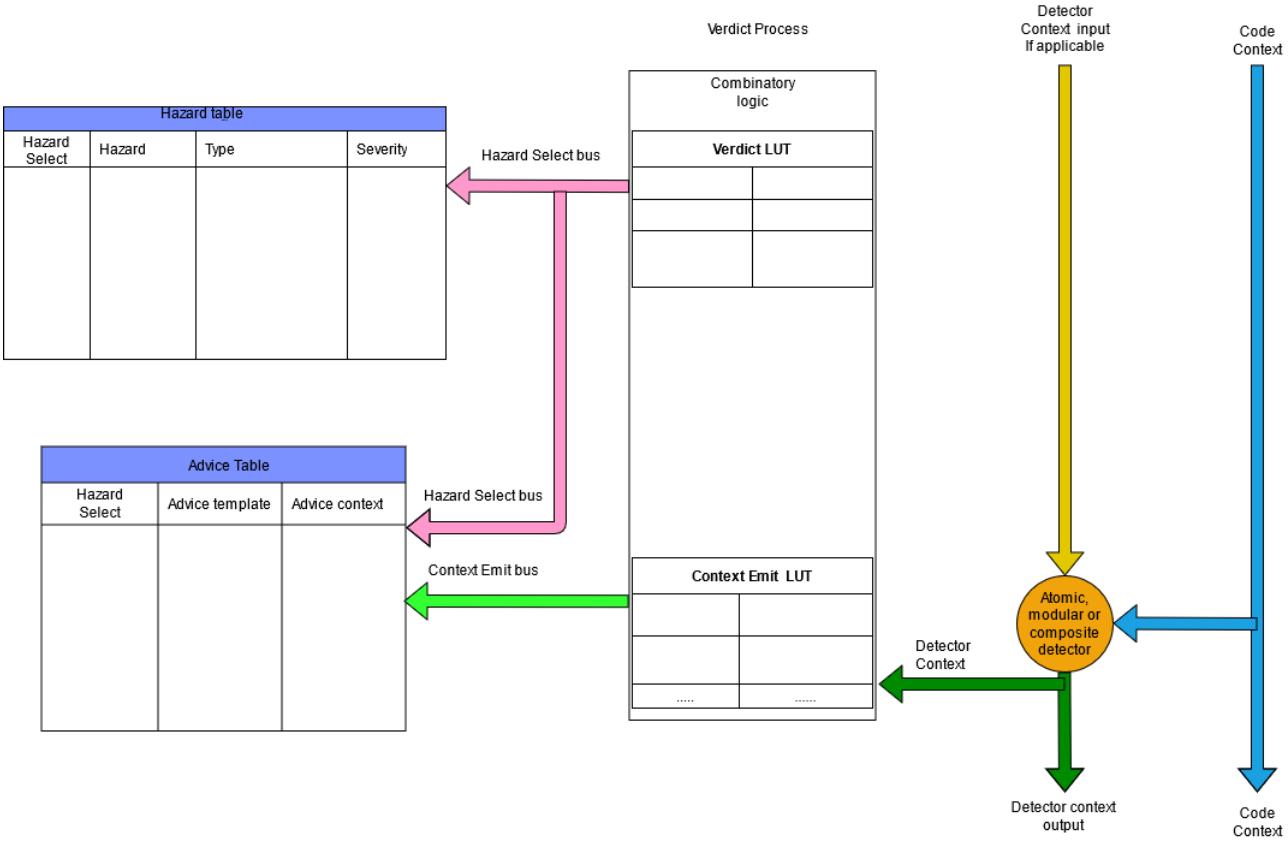

Figure 11: Detector type A

The atomic detectors are detectors that are preliminary used for single microsteps, like add method or remove method. The verdict LUT and the context emit LUT can be found in this appendix. The advice table can be found in Appendix C. The hazard tables can be found in section 7.2.2 in this document.

.2. LUTs FOR DETECTORS FOR SINGLE MICROSTEPS

.2.1. ADD CLASS

A preliminary notice, "!alias" represents an empty list. "alias" represents a list with one or more items.
Tables for the following hazard AC-H1, Add Class Hazard 1.

Verdict LUT

Detector output	Hazard
ClassInPackage	**AC-H1**

Context Emit LUT

Detector output	Context
ClassAlreadyInPackage	**<ClassAlreadyInPackage?>**

Table 17: Verdict LUT and Context emit LUT for Add Class hazard AC-H1

.2.2. REMOVE CLASS

Tables for the following hazard RC-H1-1, Remove Class Hazard 1-1.

Verdict LUT

Detector output	Hazard
findFieldInstantiationToClass	**RC-H1-1**

Context Emit LUT

Detector output	Context
findFieldInstantiationToClass	**<findFieldInstantiationToClass?>** **<iClass2BRemoved?>**

Table 18: Verdict LUT and Context emit LUT for Remove Class hazard RC-H1-1

Tables for the following hazard RC-H1-2, Remove Class Hazard 1-2.

Verdict LUT

Detector output	Hazard
FindFieldReferenceToClass	**RC-H1-2**

Context Emit LUT

Detector output	Context
FindFieldReferenceToClass	**<FindFieldReferenceToClass?>**

Table 19: Verdict LUT and Context emit LUT for Remove Class hazard RC-H1-2

Tables for the following hazard RC-H1-3, Remove Class Hazard 1-3.

Verdict LUT

Detector output	Hazard
AllCallersToMyClass	**RC-H1-3**

Context Emit LUT

Detector output	Context
AllCallersToMyClass	**< AllCallersToMyClass?>**

Table 20: Verdict LUT and Context emit LUT for Remove Class hazard RC-H1-3

Tables for the following hazard RC-H2-1, Remove Class Hazard 2-1.

Verdict LUT

Detector output	Hazard
IfEmptyNoSuperclassInTree	**RC-H2-1**

Context Emit LUT

Detector output	Context
IfEmptyNoSuperclassInTree	**<IfEmptyNoSuperclassInTree?>**

Table 21: Verdict LUT and Context emit LUT for Remove Class hazard RC-H2-1

Tables for the following hazard RC-H2-2, Remove Class Hazard 2-2.

Verdict LUT

Detector output	Hazard
IfEmptyNoSubclassInTree	**RC-H2-2**

Context Emit LUT

Detector output	Context
IfEmptyNoSubclassInTree	**<IfEmptyNoSubclassInTree?>**

Table 22: Verdict LUT and Context emit LUT for Remove Class hazard RC-H2-1

Tables for the following hazard RC-H3, Remove Class Hazard 3.

Verdict LUT

Detector output	Hazard
ListOfCallersToStaticMethodInStaticInnerClass	**RC-H3**

Context Emit LUT

Detector output	Context
ListOfCallersToStaticMethodInStaticInnerClass	**<ListOfCallersToStaticMethodInStaticInnerClass?>**

Table 23: Verdict LUT and Context emit LUT for Remove Class hazard RC-H3

Tables for the following hazard RC-H4, Remove Class Hazard 4.

Verdict LUT

Detector output	Hazard
FindClassAbstractModifier	**RC-H4**

Context Emit LUT

Detector output	Context
FindClassAbstractModifier	**<FindClassAbstractModifier?>**

Table 24: Verdict LUT and Context emit LUT for Remove Class hazard RC-H2-1

.2.3. ADD ATTRIBUTE

Tables for the following hazard AA-H1, Add Attribute Hazard 1.

Verdict LUT

Detector output	Hazard
FindFieldWithNameInClass	**AA-H1**

Context Emit LUT

Detector output	Context
FindFieldWithNameInClass	**<FindFieldWithNameInClass?>**

Table 25: Verdict LUT and Context emit LUT for Add Attribute hazard AA-H1

Tables for the following hazard AA-H2, Add Attribute Hazard 2.

Verdict LUT

Detector output	Hazard
ListAssignmentsOfFieldsInClassAA	**AA-H2**

Context Emit LUT

Detector output	Context
ListAssignmentsOfFieldsInClassAA	**<ListAssignmentsOfFieldsInClassAA?>**

Table 26: Verdict LUT and Context emit LUT for Add Attribute hazard AA-H2

Tables for the following hazard AA-H3, Add Attribute Hazard 3.

Verdict LUT

Detector output	Hazard
AssignFieldsInSupOfClassAA	**AA-H3**

Context Emit LUT

Detector output	Context
AssignFieldsInSupOfClassAA	**<AssignFieldsInSupOfClassAA?>**

Table 27: Verdict LUT and Context emit LUT for Add Attribute hazard AA-H3

.2.4. Remove Attribute

Tables for the following hazard RA-H1, Remove Attribute Hazard 1.

Verdict LUT

Detector output	Hazard
AssignToOldAttrRef	**RA-H1**

Context Emit LUT

Detector output	Context
AssignToOldAttrRef	**<AssignToOldAttrRef?>**

Table 28: Verdict LUT and Context emit LUT for Remove Attribute hazard RA-H1

Tables for the following hazard RA-H2, Remote Attribute Hazard 2.

Verdict LUT

Detector output	Hazard
ListAssignFieldsInClassRA	**RA-H2**

Context Emit LUT

Detector output	Context
ListAssignFieldsInClassRA	**<ListAssignFieldsInClassRA?>**

Table 29: Verdict LUT and Context emit LUT for Remove Attribute hazard RA-H2

Tables for the following hazard RA-H3, Remove Attribute Hazard 3.

Verdict LUT

Detector output	Hazard
AssignFieldsInSupClassRA	**RA-H3**

Context Emit LUT

Detector output	Context
AssignFieldsInSupClassRA	**< AssignFieldsInSupClassRA?>**

Table 30: Verdict LUT and Context emit LUT for Remove Attribute hazard RA-H3

.2.5. ADD METHOD

Tables for the following hazard AM-H1, Add Method Hazard 1.

Verdict LUT

Detector output	Hazard
MethodAlreadyDefinedInClassAM	**AM-H1**

Context Emit LUT

Detector output	Context
MethodAlreadyDefinedInClassAM	**<MethodAlreadyDefinedInClassAM?>**

Table 31: Verdict LUT and Context emit LUT for Add Method hazard AM-H1

Tables for the following hazard AM-H2, Add Method Hazard 2.

Verdict LUT

Detector output	Hazard
CallSubToSigYinSuperNoKeyAM	**AM-H2**

Context Emit LUT

Detector output	Context
CallSubToSiginSupNoKeyAM	**<CallSubToSiginSupNoKeyAM?>**

Table 32: Verdict LUT and Context emit LUT for Add Method hazard AM-H2

.2.6. Remove Method

Tables for the following hazard RM-H1, Remove Method Hazard 1.

Verdict LUT

Detector output	Hazard
FindCallersToSignature2BRemoved	**RM-H1**

Context Emit LUT

Detector output	Context
FindCallersToSignature2BRemoved	**<FindCallersToSignature2BRemoved?>**

Table 33: Verdict LUT and Context emit LUT for Remove Method hazard RA-H1

Tables for the following hazard RM-H2, Remove Method Hazard 2.

Verdict LUT

Detector output	Hazard
CallSubToSiginSupNoKeyRM	**AM-H3**

Context Emit LUT

Detector output	Context
CallSubToSiginSupNoKeyRM	**<CallSubToSiginSupNoKeyRM?>**

Table 34: Verdict LUT and Context emit LUT for Remove Method hazard RM-H2

Tables for the following hazard RM-H3, Remove Method Hazard 3.

Verdict LUT

Detector output	Hazard
MethodStaticDefaultDefInInterfaceRM	**RM-H3**

Context Emit LUT

Detector output	Context
MethodStaticDefaultDefInInterfaceRM	**<MethodStaticDefaultDefInInterfaceRM?>**

Table 35: Verdict LUT and Context emit LUT for Remove Method hazard RM-H3

.3. Add Interface

Tables for the following hazard AI-H1, Add Interface Hazard 1.

Verdict LUT

Detector output	Hazard
NewInterfaceAlreadyDefined	**AI-H1**

Context Emit LUT

Detector output	Context
NewInterfaceAlreadyDefined	**<NewInterfaceAlreadyDefined?>**

Table 36: Verdict LUT and Context emit LUT for Add Interface hazard AI-H1

.4. Remove Interface

Tables for the following hazard RI-H1, Remove Interface Hazard 1.

Verdict LUT

Detector output	Hazard
ClassThatImplInterfaceToRemove	**RI-H1**

Context Emit LUT

Detector output	Context
ClassThatImplInterfaceToRemove	**<ClassThatImplInterfaceToRemove?>**

Table 37: Verdict LUT and Context emit LUT for Remove Method hazard RI-H1

Tables for the following hazard RI-H2, Remove Interface Hazard 2.

Verdict LUT

Detector output	Hazard
InterfaceThatImplInterfaceToRemove	**RI-H2**

Context Emit LUT

Detector output	Context
InterfaceThatImplInterfaceToRemove	**<InterfaceThatImplInterfaceToRemove?>**

Table 38: Verdict LUT and Context emit LUT for Remove Method hazard RI-H2

Tables for the following hazard RI-H3, Remove Interface Hazard 3.

Verdict LUT

Detector output	Hazard
StaticInterfaceImplInUse	**RI-H3**

Context Emit LUT

Detector output	Context
StaticInterfaceImplInUse	**<StaticInterfaceImplInUse?>**

Table 39: Verdict LUT and Context emit LUT for Remove Interface hazard RI-H3

.5. Add Parameter

Tables for the following hazard AP-H1, Add Parameter Hazard 1.

Verdict LUT

Detector output	Hazard
FindCallersToSignatureWithNewParam	**AP-H1**

Context Emit LUT

Detector output	Context
FindCallersToSignatureWithNewParam	**<FindCallersToSignatureWithNewParam?>**

Table 40: Verdict LUT and Context emit LUT for Add Parameter hazard AP-H1

Tables for the following hazard AP-H2, Add Parameter Hazard 2.

Verdict LUT

Detector output	Hazard
MethodAlreadyDefinedInClassAP	**AP-H2**

Context Emit LUT

Detector output	Context
MethodAlreadyDefinedInClassAP	**<MethodAlreadyDefinedInClassAP?>**

Table 41: Verdict LUT and Context emit LUT for Add Parameter hazard AP-H1

Tables for the following hazard AP-H3 Add Parameter Hazard 3.

Verdict LUT

Detector output	Hazard
CallSubToSiginSupNoKeyAP	**AP-H3**

Context Emit LUT

Detector output	Context
CallSubToSiginSupNoKeyAP	**<CallSubToSiginSupNoKeyAP?>**

Table 42: Verdict LUT and Context emit LUT for Add Parameter hazard AP-H3

Tables for the following hazard AP-H4, Add Parameter Hazard 4.

Verdict LUT

Detector output	Hazard
MethodStatDefDefInInterfAP	**AP-H4**

Context Emit LUT

Detector output	Context
MethodStatDefDefInInterfAP	**<MethodStatDefDefInInterfAP?>**

Table 43: Verdict LUT and Context emit LUT for Add Parameter hazard AP-H4

Tables for the following hazard AP-H5 Add Parameter Hazard 5.

Verdict LUT

Detector output	Hazard
NameIsAlreadyPresentInParamListAP	**AP-H5**

Context Emit LUT

Detector output	Context
NameIsAlreadyPresentInParamListAP	**<iNameNew?>**

Table 44: Verdict LUT and Context emit LUT for Add Parameter hazard AP-H5

Tables for the following hazard AP-H6 Add Parameter Hazard 6.

Verdict LUT

Detector output	Hazard
ParamInUseByFieldAP	**AP-H6**

Context Emit LUT

Detector output	Context
ParamInUseByFieldAP	**<iNameNew?>**

Table 45: Verdict LUT and Context emit LUT for Add Parameter hazard AP-H6

.6. Remove Parameter

Tables for the following hazard RP-H1, Remove Parameter Hazard 1.

Verdict LUT

Detector output	Hazard
FindCallersToSignatureWithRemovedParamRP	**RP-H1**

Context Emit LUT

Detector output	Context
FindCallersToSignatureWithRemovedParamRP	**<FindCallersToSignatureWithRemovedParamRP?>**

Table 46: Verdict LUT and Context emit LUT for Remove Parameter hazard RP-H1

Tables for the following hazard RP-H2, Remove Parameter Hazard 2.

Verdict LUT

Detector output	Hazard
MethodAlreadyDefinedInClassRP	**RP-H2**

Context Emit LUT

Detector output	Context
MethodAlreadyDefinedInClassRP	**<MethodAlreadyDefinedInClassRP?>**

Table 47: Verdict LUT and Context emit LUT for Remove Parameter hazard RP-H1

Tables for the following hazard RP-H3 Remove Parameter Hazard 3.

Verdict LUT

Detector output	Hazard
CallSubToSiginSupNoKeyRP	**RP-H3**

Context Emit LUT

Detector output	Context
CallSubToSiginSupNoKeyRP	**<CallSubToSiginSupNoKeyRP?>**

Table 48: Verdict LUT and Context emit LUT for Remove Parameter hazard RP-H3

Tables for the following hazard RP-H4, Remove Parameter Hazard 4.

Verdict LUT

Detector output	Hazard
MethodStatDefDefInInterfRP	**RP-H4**

Context Emit LUT

Detector output	Context
MethodStatDefDefInInterfRP	**<MethodStatDefDefInInterfRP?>**

Table 49: Verdict LUT and Context emit LUT for Remove Parameter hazard RP-H4

Tables for the following hazard RP-H5 Remove Parameter Hazard 5.

Verdict LUT

Detector output	Hazard
ParamInUseInsideMethodRP	**RP-H5**

Context Emit LUT

Detector output	Context
ParamInUseInsideMethodRP	**<ParamInUseInsideMethodRP?>** **<iOldMethod?>**

Table 50: Verdict LUT and Context emit LUT for Remove Parameter hazard RP-H5

APPENDIX E - COMPOSITE DETECTOR RECIPES FOR CASCADED MICROSTEPS

The detector recipes comprehend filling tables with the correct information and providing a hazardous code presence detection algorithm designed with the query language.
For all composite detectors, detector type B is used, see figure 12.

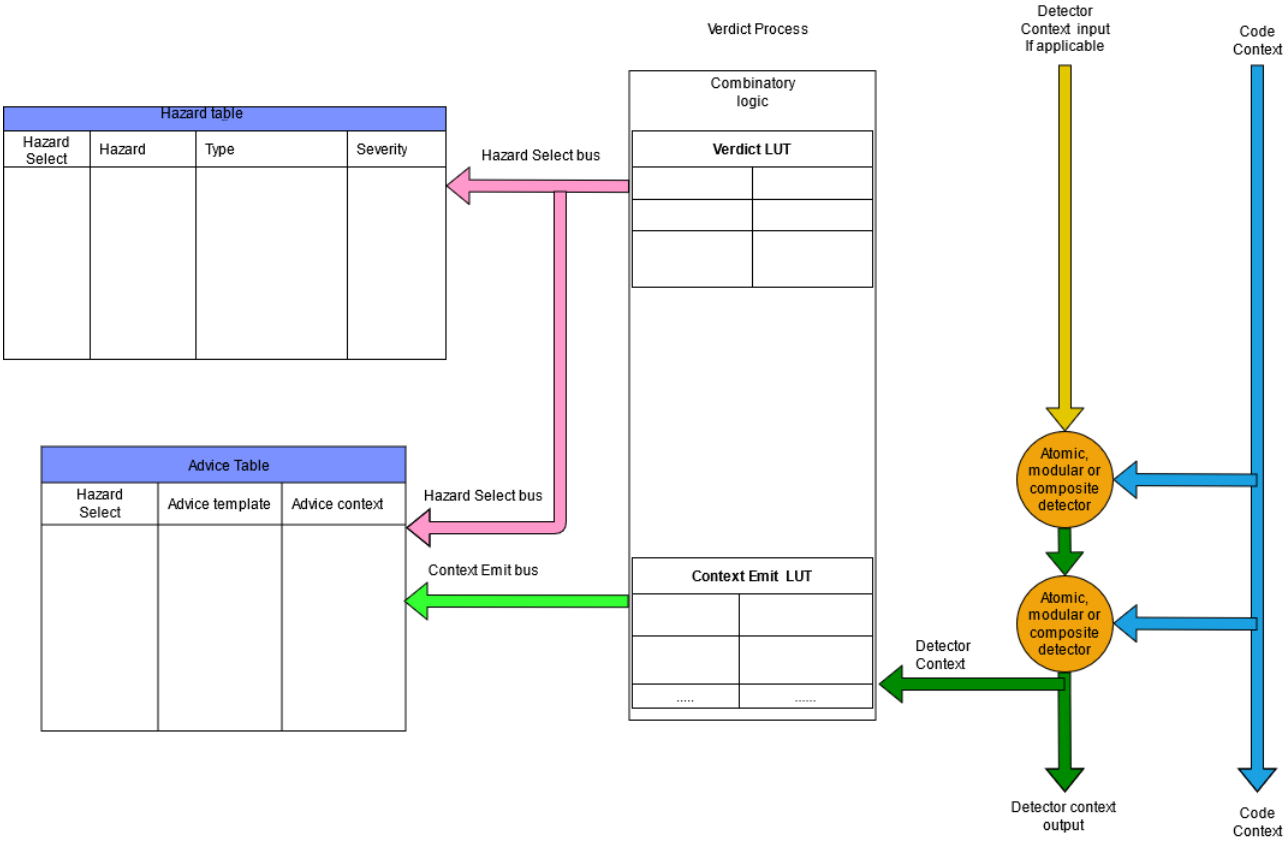

Figure 12: Detector type B

The composite detectors are detectors that are used for cascaded microsteps, like rename method, or rename class. Rename microstep consist out of a cascaded Remove- and Add microstep. The verdict LUT and the context emit LUT can be found in this appendix. The advice table can be found in Appendix C. The hazard tables can be found in section 7.2.2 in this document.

.7. LUTs FOR COMPOSITE DETECTORS FOR CASCADED MICROSTEPS

A preliminary notice, "!alias" represents an empty list. "alias" represents a list with one or more items.

.7.1. RENAME CLASS

Tables for the following hazard RNC-H1-1, Rename Class Hazard 1-1. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
findFieldInstantiationToClass	**RNC-H1-1**

Context Emit LUT

Detector output	Context
findFieldInstantiationToClass	**<findFieldInstantiationToClass?> <iClass2BRemoved?>**

Table 51: Verdict LUT and Context emit LUT for Rename Class hazard RNC-H1-1

Tables for the following hazard RNC-H1-2, Rename Class Hazard 1-2. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
FindFieldReferenceToClass	**RNC-H1-2**

Context Emit LUT

Detector output	Context
FindFieldReferenceToClass	**<FindFieldReferenceToClass?>**

Table 52: Verdict LUT and Context emit LUT for Rename Class hazard RNC-H1-2

Tables for the following hazard RNC-H1-3, Rename Class Hazard 1-3. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
AllCallersToMyClass	**RNC-H1-3**

Context Emit LUT

Detector output	Context
AllCallersToMyClass	**<AllCallersToMyClass?>**

Table 53: Verdict LUT and Context emit LUT for Rename Class hazard RNC-H1-3

Tables for the following hazard RNC-H2-1, Rename Class Hazard 2-1. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
IfEmptyNoSuperclassInTree	**RNC-H2-1**

Context Emit LUT

Detector output	Context
IfEmptyNoSuperclassInTree	**<IfEmptyNoSuperclassInTree?>**

Table 54: Verdict LUT and Context emit LUT for Rename Class hazard RNC-H2-1

Tables for the following hazard RNC-H2-2, Rename Class Hazard 2-2. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
IfEmptyNoSubclassInTree	**RNC-H2-2**

Context Emit LUT

Detector output	Context
IfEmptyNoSubclassInTree	**<IfEmptyNoSubclassInTree?>**

Table 55: Verdict LUT and Context emit LUT for Rename Class hazard RNC-H2-2

Tables for the following hazard RNC-H3, Rename Class Hazard 3. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
ListOfCallersToStaticMethodInStaticInnerClass	**RNC-H3**

Context Emit LUT

Detector output	Context
ListOfCallersToStaticMethodInStaticInnerClass	**<ListOfCallersToStaticMethodInStaticInnerClass?>**

Table 56: Verdict LUT and Context emit LUT for Remove Method hazard RNC-H3

Tables for the following hazard RNC-H4, Rename Class Hazard 4. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
FindClassAbstractModifier	**RNC-H4**

Context Emit LUT

Detector output	Context
FindClassAbstractModifier	**<FindClassAbstractModifier?>**

Table 57: Verdict LUT and Context emit LUT for Rename Class hazard RNC-H4

Tables for the following hazard RNC-H5, Rename Class Hazard 5. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
ClassAlreadyInPackage	**RNC-H5**

Context Emit LUT

Detector output	Context
ClassAlreadyInPackage	**<ClassAlreadyInPackage?>**

Table 58: Verdict LUT and Context emit LUT for Rename Class hazard RNC-H5

.7.2. Rename Attribute

Tables for the following hazard RNA-H1, Rename Attribute Hazard 1. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
AssignToOldAttrRef	**RNA-H1**

Context Emit LUT

Detector output	Context
StaticDefaultImpl	**<AssignToOldAttrRef?>** **<iIsAttributeToAdd?>**

Table 59: Verdict LUT and Context emit LUT for Rename Attribute hazard RNA-H1

Tables for the following hazard RNA-H2, Rename Attribute Hazard 2. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
ListAssignFieldsInClassRA	**RNA-H2**

Context Emit LUT

Detector output	Context
StaticDefaultImpl	**<iIsAttributeToAdd?>** **<ListAssignFieldsInClassRA?>**

Table 60: Verdict LUT and Context emit LUT for Rename Attribute hazard RNA-H2

Tables for the following hazard RNA-H3, Rename Attribute Hazard 3. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
AssignFieldsInSupClassRA	**RNA-H3**

Context Emit LUT

Detector output	Context
StaticDefaultImpl	**<iIsAttributeToAdd?>** **<AssignFieldsInSupClassRA?>**

Table 61: Verdict LUT and Context emit LUT for Rename Attribute hazard RNA-H3

Tables for the following hazard RNA-H4, Rename Attribute Hazard 4. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
FindFieldWithNameInClass	**RNA-H4**

Context Emit LUT

Detector output	Context
FindFieldWithNameInClass	**<iIsAttributeToAdd?>** **<FindFieldWithNameInClass?>**

Table 62: Verdict LUT and Context emit LUT for Rename Attribute hazard RNA-H4

.7.3. Rename Method

Tables for the following hazard RNM-H1, Rename Attribute Hazard 1. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
FindCallersToSignature2BRemoved?	**RNM-H1**

Context Emit LUT

Detector output	Context
FindCallersToSignature2BRemoved?	**< FindCallersToSignature2BRemoved?> <iNewSignature?>**

Table 63: Verdict LUT and Context emit LUT for Rename Method hazard RNM-H1

Tables for the following hazard RNM-H2, Rename Method Hazard 2. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
MethodAlreadyDefinedInClass	**RNM-H2**

Context Emit LUT

Detector output	Context
StaticDefaultImpl	**<iNewSignature?> <MethodAlreadyDefinedInClass?>**

Table 64: Verdict LUT and Context emit LUT for Rename Method hazard RNM-H2

Tables for the following hazard RNM-H3, Rename Method Hazard 3. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
CallSubToSiginSupNoKeyAM	**RNM-H3**

Context Emit LUT

Detector output	Context
StaticDefaultImpl	**<iNewSignature?> <CallSubToSiginSupNoKeyAM?>**

Table 65: Verdict LUT and Context emit LUT for Rename Method hazard RNM-H3

Tables for the following hazard RNM-H4, Rename Method Hazard 4. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
MethodStaticDefaultDefInInterface	**RNM-H4**

Context Emit LUT

Detector output	Context
MethodStaticDefaultDefInInterface	**<MethodStaticDefaultDefInInterface?>**

Table 66: Verdict LUT and Context emit LUT for Remove Method hazard RNM-H4

.7.4. RENAME INTERFACE

Tables for the following hazard RNI-H1, Rename Interface Hazard 1. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
ClassThatImplInterfaceToRemove	**RNI-H1**

Context Emit LUT

Detector output	Context
ClassThatImplInterfaceToRemove	**<iInterFace2Remove?>** **< ClassThatImplInterfaceToRemove?>** **<iInterFace2Remove?>**

Table 67: Verdict LUT and Context emit LUT for Rename Interface hazard RNI-H1

Tables for the following hazard RNI-H2, Rename Interface Hazard 2. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
InterfaceThatImplInterfaceToRemove	**RNI-H2**

Context Emit LUT

Detector output	Context
InterfaceThatImplInterfaceToRemove	**<InterfaceThatImplInterfaceToRemove?>** **<InterfaceThatImplInterfaceToRemove?>**

Table 68: Verdict LUT and Context emit LUT for Rename Interface hazard RNI-H1

Tables for the following hazard RNI-H3, Rename Interface Hazard 3. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
StaticInterfaceImplInUse	**RNI-H3**

Context Emit LUT

Detector output	Context
StaticInterfaceImplInUse	**<StaticInterfaceImplInUse?>**

Table 69: Verdict LUT and Context emit LUT for Rename Interface hazard RNI-H3

Tables for the following hazard RNI-H4, Rename Interface Hazard 4. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
NewInterfaceAlreadyDefined	**RNI-H4**

Context Emit LUT

Detector output	Context
NewInterfaceAlreadyDefined	**<NewInterfaceAlreadyDefined?>**

Table 70: Verdict LUT and Context emit LUT for Rename Interface hazard RNI-H4

.7.5. Rename Parameter

Tables for the following hazard RNP-H1, Rename Parameter Hazard 1. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
NameIsAlreadyPresentInParamListAP	**RNP-H1**

Context Emit LUT

Detector output	Context
NameIsAlreadyPresentInParamListAP	**<iNameNew?>**

Table 71: Verdict LUT and Context emit LUT for Rename Parameter hazard RNP-H1

Tables for the following hazard RNP-H2, Rename Parameter Hazard 2. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
ParamInUseByFieldAP	**RNP-H2**

Context Emit LUT

Detector output	Context
ParamInUseByFieldAP	**<iNameNew?>**

Table 72: Verdict LUT and Context emit LUT for Rename Parameter hazard RNP-H2

Tables for the following hazard RNP-H3, Rename Parameter Hazard 3. Which is a cascade of the Remove- and Add microsteps.

Verdict LUT

Detector output	Hazard
ParamInUseInsideMethodRP	**RNP-H3**

Context Emit LUT

Detector output	Context
ParamInUseInsideMethodRP	**<iNameNew?>**

Table 73: Verdict LUT and Context emit LUT for Rename Parameter hazard RNP-H3

APPENDIX F - AN IDEA FOR AN ENHANCED ADVISE MECHANISM

The advice mechanism should be able to investigate facts indepent of the detectors. The detectors deliver the lead(nodes), and the advice mechanism can investigate by using the query language as can be seen in figure 13. In this example, the detectors deliver a list callers (ast nodes), but for the advice more information is necessary, like in which method is that caller located? Or even, on which line in the source code.

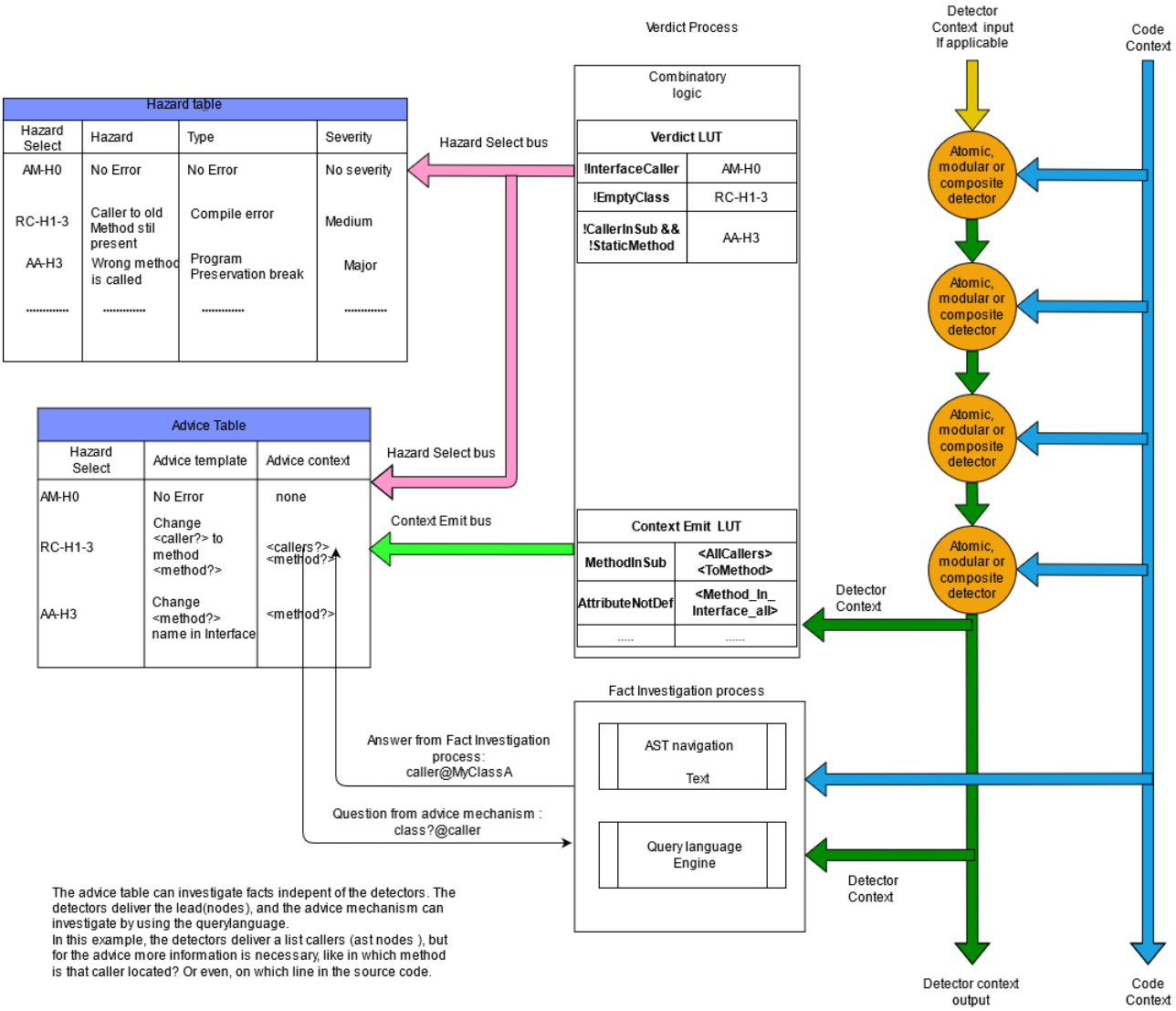

Figure 13: An enhanced, and further decoupled, advise mechanism

ACRONYMS

AST Abstract Syntax Tree. iii, ix, 1–3, 14, 26–40, 52, 61, 62, 68, 72, 73, 76, 77

CCA Code Context Advice. ii–v

CCP Code Context Property. iii, iv

CCPD Code Context Property Detector. iii

IDE Integrated Development Environment. 5

LUT Look Up Table. iv, x–xl, 63, 66–68, 70–72, 74, 75

RAG Refactoring Advice Graph. ii–v, 5, 7, 8